**What is OWASP(Open Web Application Security Project)?**

At OWASP, you’ll find descriptions of the most common vulnerabilities that you should avoid in your applications. Let's take a few minutes and discuss these theoretically before diving into the next chapters, where you’ll start to apply concepts from Spring Security. Among the common vulnerabilities that you should be aware of, you’ll find these:

* Broken authentication
* Session fixation
* Cross-site scripting (XSS)
* Cross-site request forgery (CSRF)
* Injections
* Sensitive data exposure
* Lack of method access control
* Using dependencies with known vulnerabilities

**What is Session Fixation?**

Session fixation vulnerability is a more specific, high-severity weakness of a web application. If present, it permits an attacker to impersonate a valid user by reusing a previously generated session ID. This vulnerability can happen if, during the authentication process, the web application does not assign a unique session ID. This can potentially lead to the reuse of existing session IDs. Exploiting this vulnerability consists of obtaining a valid session ID and making the intended victim's browser use it.

**What is Cross-Site Scripting (XSS)?**

Cross-site scripting, also referred to as XSS, allows the injection of client-side scripts into web services exposed by the server, thereby permitting other users to run these. Before being used or even stored, you should properly “sanitize” the request to avoid undesired executions of foreign scripts.

**What is CSRF attach?**

Cross-site request forgery (CSRF) vulnerabilities are also common in web applications. CSRF attacks assume that a URL that calls an action on a specific server can be extracted and reused from outside the application ([figure 1.8](https://cdn2.percipio.com/1691142909.c0a6b0d0200611018163610725529c7d704e7659/eod/books/154195/OEBPS/section-14-13.xhtml#fig1-8)). If the server trusts the execution without doing any check on the origin of the request, one could execute it from any other place. Through CSRF, an attacker can make a user execute undesired actions on a server by hiding the actions. Usually, with this vulnerability, the attacker targets actions that change data in the system.

**How does basic authentication flow work?**

The most straightforward but least desirable approach as a practical solution is to use HTTP Basic for endpoint authentication. While this approach is direct to understand and generally used with the first theoretical examples of authentication, it does have leaks that you want to avoid. For example, using HTTP Basic implies sending the credentials with each call. As you’ll see in chapter 2, credentials aren't encrypted. The browser sends the username and the passwords as a Base64 encoding. This way, the credentials are left available on the network in the header of each endpoint call. Also, assuming that the credentials represent the user that's logged in, you don't want the user to type their credentials for every request. You also don't want to have to store the credentials on client side. This practice is not advisable.

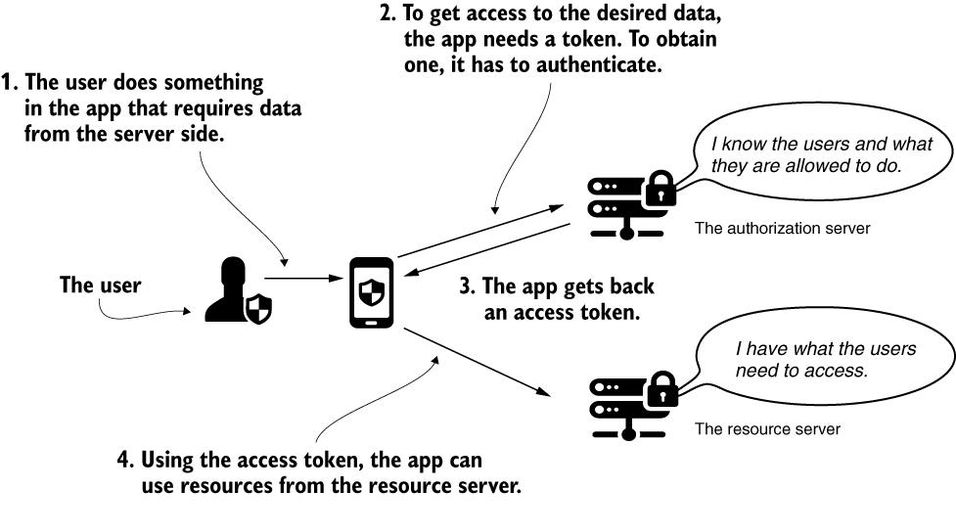
**Can you explain OAuth2 Authorization flow?**

We certainly want to find a solution to avoid resending credentials for each of the requests to the backend and store these on the client side.

The OAuth 2 flow offers a better way to implement authentication and authorization in these cases.

In OAuth 2 authorization flow. Step by step, the following happens:

1. The user accesses a use case in the application (also known as the client). The application needs to call a resource in the backend.
2. To be able to call the resource, the application first has to obtain an access token, so it calls the authorization server to get the token. In the request, it sends the user credentials or a refresh token, in some cases.
3. If the credentials or the refresh token are correct, the authorization server returns a (new) access token to the client.
4. The header of the request to the resource server uses the access token when calling the needed resources.



A token is like an access card you use inside an office building. As a visitor, you first visit the front desk, where you receive an access card after identifying yourself. The access card can open some of the doors, but not necessarily all. Based on your identity, you can access precisely the doors that you’re allowed to and no more. The same happens with an access token. After authentication, the caller is provided with a token, and based on that, they can access the resources for which they have privileges.

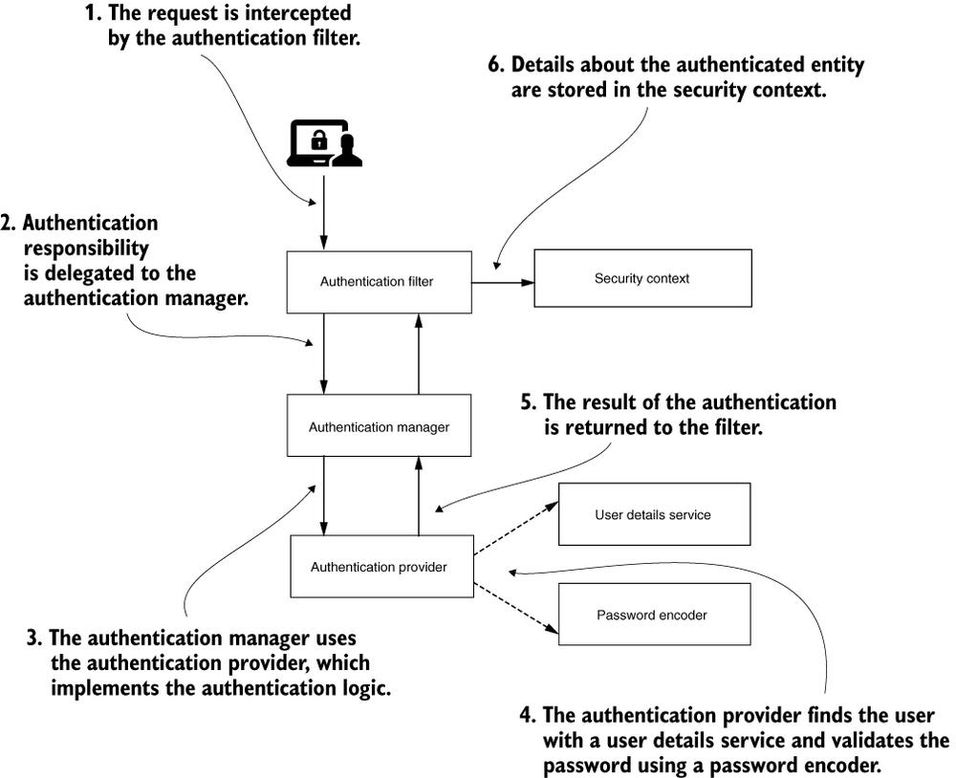
A token has a fixed lifetime, usually being short-lived. When a token expires, the app needs to obtain a new one. If needed, the server can disqualify the token earlier than its expiration time. The following lists some of the advantages of this flow:

* The client doesn't have to store the user credentials. The access token and, eventually, the refresh token are the only access details you need to save.
* The application doesn't expose the user credentials, which are often on the network.
* If someone intercepts a token, you can disqualify the token without needing to invalidate the user credentials.
* A token can be used by a third entity to access resources on the user's behalf, without having to impersonate the user. Of course, an attacker can steal the token in this case. But because the token usually has a limited lifespan, the timeframe in which one can use this vulnerability is limited.

**How does component autoscan happen in spring boot application?**

Spring Boot scans for components only in the package (and its subpackages) that contains the class annotated with @SpringBootApplication. If you annotate classes with any of the stereotype components in Spring outside of the main package, you must explicitly declare the location using the @ComponentScan annotation.

**Can you explain the main components in authentication process in Spring security and relationship among them?**

Larger View

**How do you configure HTTPS for your application?**

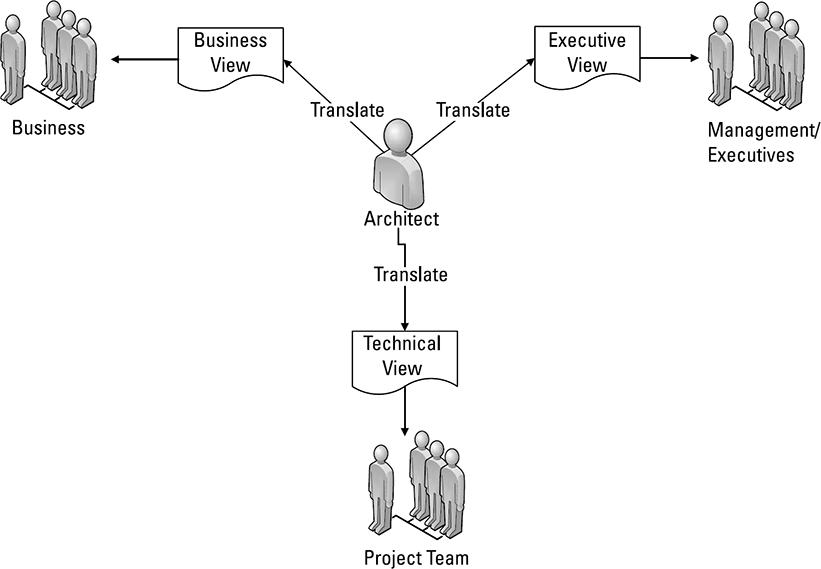
There are several patterns to configure HTTPS in a system. In some cases, developers configure HTTPS at the application level; in others, they might use a service mesh or they could choose to set HTTPS at the infrastructure level. With Spring Boot, you can easily enable HTTPS at the application level.

In any of these configuration scenarios, you need a certificate signed by a certification authority (CA). Using this certificate, the client that calls the endpoint knows whether the response comes from the authentication server and that nobody intercepted the communication. You can buy such a certificate, but you have to renew it. If you only need to configure HTTPS to test your application, you can generate a self-signed certificate using a tool like OpenSSL. Let's generate our self-signed certificate and then configure it in the project:

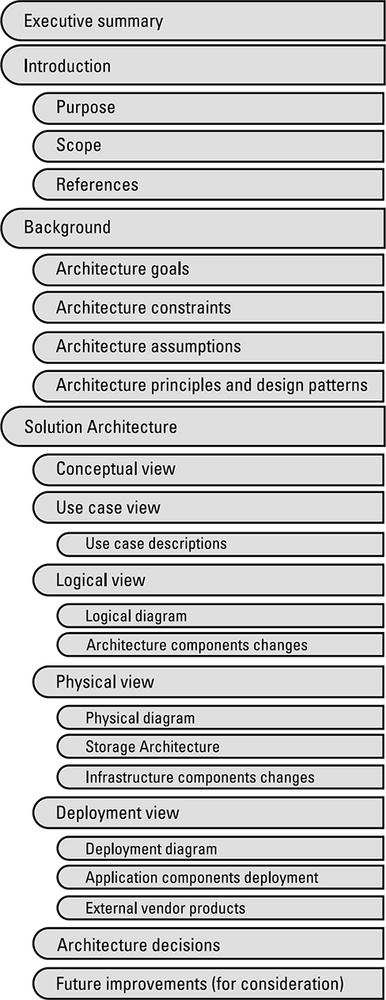
**What is difference between Designer and Architect?**

*The designer is concerned with what happens when a user presses a button and the architect is concerned with what happens when ten thousand users press a button.*

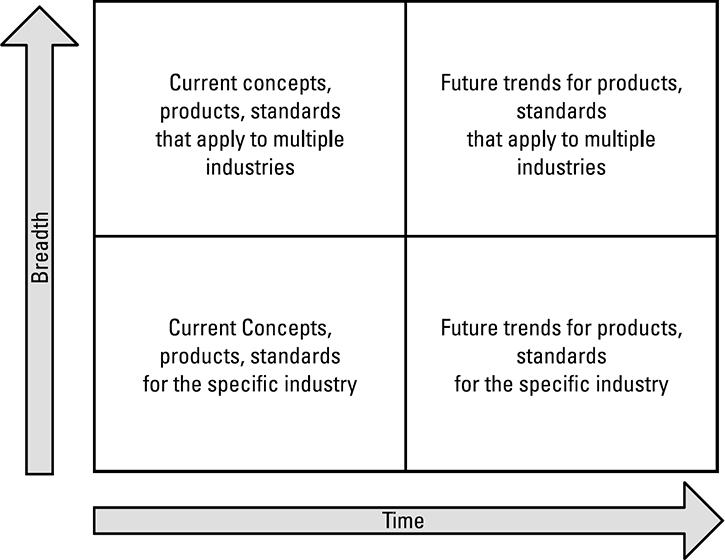
**Architect as the interface between technical, management, and business teams?**



**Example of a solution architecture document structure?**



**Industry knowledge dimensions for IT architects?**

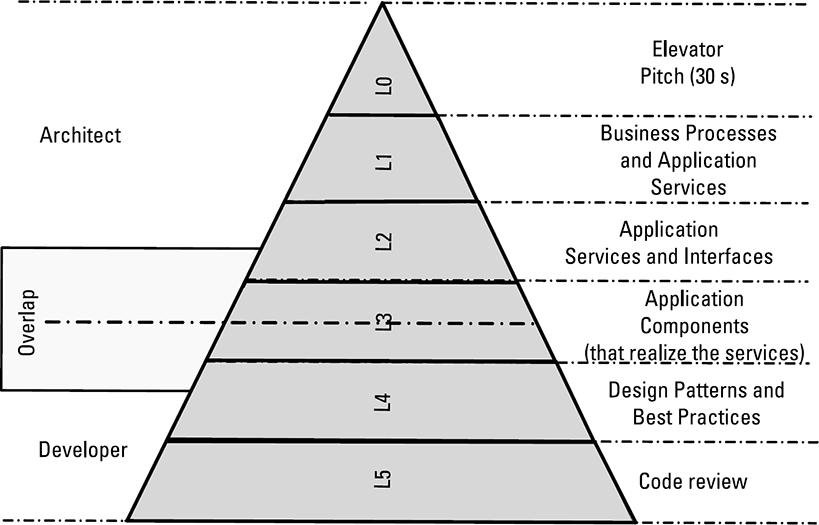


**Can you explain example criteria for an architectural decision?**

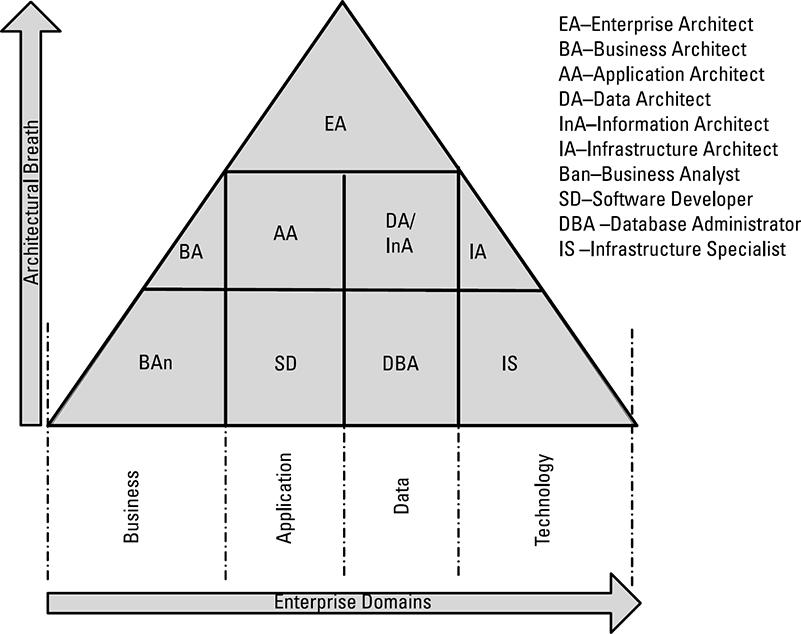
example criteria for an architectural decision.

|  | **Option 1: Microservices Deployed in the Docker Container** | **Option 2: SOAP Web Service Deployed in JBoss Fuse** |
| --- | --- | --- |
| **Business strategies****Business risks****Compliance Budget****Scope****Quality** **Schedule** | This is the first implementation of microservices in the organization; best practices and patterns need to be developed. | Schedule will not be impacted as the organization has already best practices and patterns for SOAP implementations in place. |
| **Delivery risk****Technical risk****IT operational cost****Business operational cost****Technical risk** **Training** | Requires training as the development team does not have the necessary knowledge | No extra training is required as the development team has extensive experience JBoss Fuse. |
| **Solution direction** | This option is in line with the mid-tier strategy closing the gap to achieve the target state. | This option maintains the status quo and does not close the gap to achieve the target state |
| **Organizational compatibility** **Availability Maintainability** | Easier to maintain as a defect or a functionality change affects a limited number of services. | More difficult to maintain as all the services are incorporated in one EAR file which has to be redeployed every time a change is implemented |
| **Manageability Performance** | Microservices are light services that can easily handle an increase in workload. | More work is required to properly adjust the performance of the SOAP services. |
| **Reliability Recoverability****Component Disaster****Accessibility Adaptability** **Interoperability** |  |  |
| **Scalability** | Microservices scale very well, especially when deployed in a virtualized/cloud environment. New instances can be started in seconds | Not so scalable and flexible, formal verbose contracts, state full with bigger payload, does not scale so well. |
| **Portability** **Extensibility** | The functionality can be easily extended as each piece of functionality is encapsulated in a separate service. | Extending the SOAP service and adding more methods requires redeployment of the whole ear file and regression testing. |
| **Assurance****Auditability** **Security** | Requires more work to secure the services including the communication between them. | SOAP has protocol extensions WS\* to ensure the security of the information transmitted. |
| **Privacy****Integrity****Credibility** **Usability** |  |  |

**Level of detail for application architecture?**



**IT roles and the EA breadth?**



**Explain some architecture-Specific Knowledge needed?**

Such knowledge comprises various architecture patterns, frameworks, and methodologies, as well as modelling languages that enable you to visually describe the models you create, and the tools that implement those languages.

**What are the different certifications for architect role?**

* Open Certified Architect: Competency-based certification (closest to the PMP one for the PM). This is applicable for enterprise, business, or IT architecture and requires the applicant to demonstrate real-world experience as opposed to the knowledge base alone (for more information, refer to <http://www3.opengroup.org/certifications/professional/open-ca>).
* TOGAF certification: Widely recognized as the best EA certification to hold as the TOGAF framework/methodology has been adopted in 80% of the Fortune 500 companies. The certification is split in two parts: the first part (the foundation), which is more suitable for those entering the profession, and the full certification, which includes scenario-based questions that try to assess the experience of the applicant [[4](https://cdn2.percipio.com/1691177485.b828d5d51e05434f454aa1494758bfc2e658cb2f/eod/books/132763/OEBPS/section-61.xhtml#ch10bib04)].
* CITA: IASA's Certified IT Architect [[5](https://cdn2.percipio.com/1691177485.b828d5d51e05434f454aa1494758bfc2e658cb2f/eod/books/132763/OEBPS/section-61.xhtml#ch10bib05)] (<http://iasaglobal.org/certifications/>). A multilevel certification program known as the Certified IT Architect that is offered at foundation, associate, specialist, and professional levels. The CITA-S Specialization credential provides focus in one of four areas: business, infrastructure, information, and software architecture.
* Zachman Certification: Integrates learned theory with real-world experience. There are three levels of certification that go from Level 1 (defined as EA associate) to Level 2 (EA professional) and Level 3 (EA instructor) [[6](https://cdn2.percipio.com/1691177485.b828d5d51e05434f454aa1494758bfc2e658cb2f/eod/books/132763/OEBPS/section-61.xhtml#ch10bib06)].
* Certified SOA Architect: Has a declared and definite emphasis on service-oriented architectures (SOAs), along with related technology solutions and infrastructure. It requires mastering the design of the service-oriented technology architecture, development, and delivery of working SOA solutions and the ability to integrate SOA solutions into general IT infrastructures. It is mostly geared towards software/application architecture.

**What is container?**

The container is roughly analogous to the VM. A major difference is that containers do not require their own full-blown OS. In fact, all containers on a single host share the host’s OS.

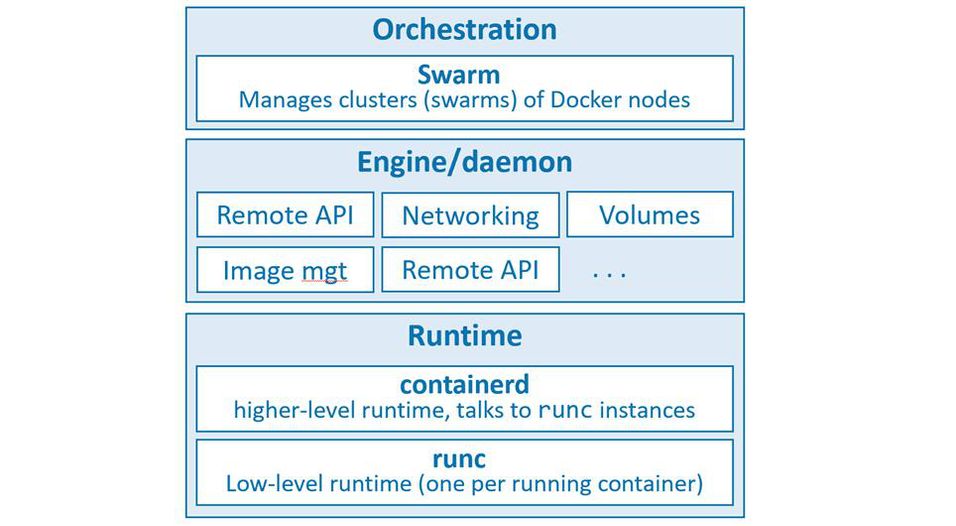
**What is containerized app?**

*A containerized app is an application running as a container.*

**What is docker?**

Docker is software that runs on Linux and Windows. **It creates, manages, and can even orchestrate containers**. The software is currently built from various tools from the Moby open-source project. Docker, Inc. is the company that created the technology and continues to create technologies and solutions that make it easier to get the code on your laptop running in the cloud.

**Can you explain docker architecture?**

.

**What are Windows Pre-Reqs for installing docker?**

Docker Desktop on Windows requires all of the following:

* 64-bit version of Windows 10 Pro/Enterprise/Education (does not work with Home edition)
* Hardware virtualization support must be enabled in your system’s BIOS
* The Hyper-V and Containers features must be enabled in Windows.

The installer can enable the Hyper-V and Containers features, but it’s your responsibility to enable hardware virtualization in your BIOS (be very careful changing anything in your system’s BIOS).

**What is a docker image?**

It’s useful to think of a Docker image as an object that contains an OS filesystem, an application, and all application dependencies. If you work in operations, it’s like a virtual machine template. A virtual machine template is essentially a stopped virtual machine. In the Docker world, an image is effectively a stopped container. If you’re a developer, you can think of an image as a class.

an image contains enough of an operating system (OS), as well as all the code and dependencies to run whatever application it’s designed for. The ubuntu image that we’ve pulled has a stripped-down version of the Ubuntu Linux filesystem, including a few of the common Ubuntu utilities. The mcr.microsoft.com/powershell:lts-nanoserver-1903 image contains a Windows Server Core OS plus PowerShell.

**What is Dockerfile?**

Dockerfile is a plain-text document that tells Docker how to build an app and dependencies into a Docker image.

**What do you mean by containerizing an app?**

Taking some application code from a remote Git repo and built it into a Docker image. You then ran a container from it. We call this “containerizing an app”.

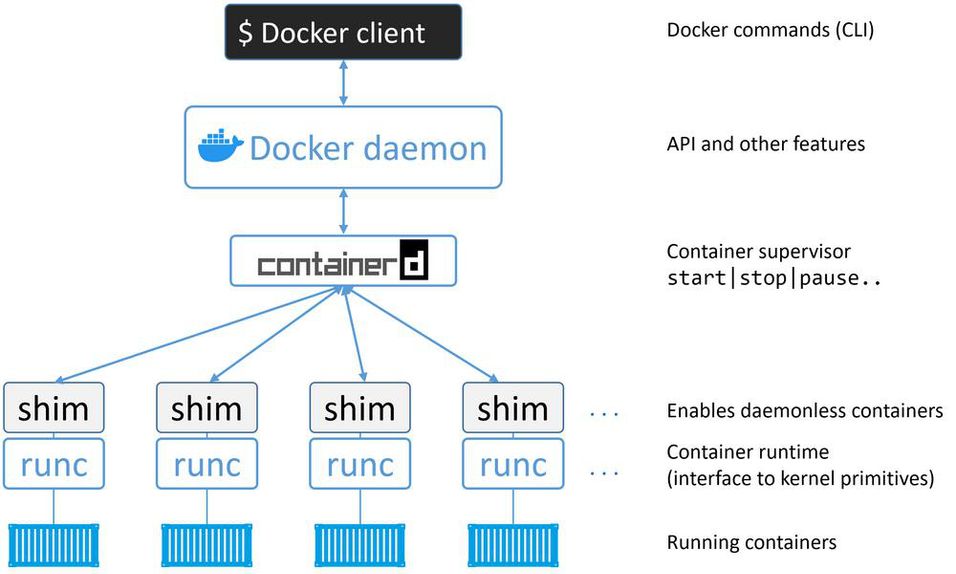
**High level view of docker engine?**

shows a high-level view.

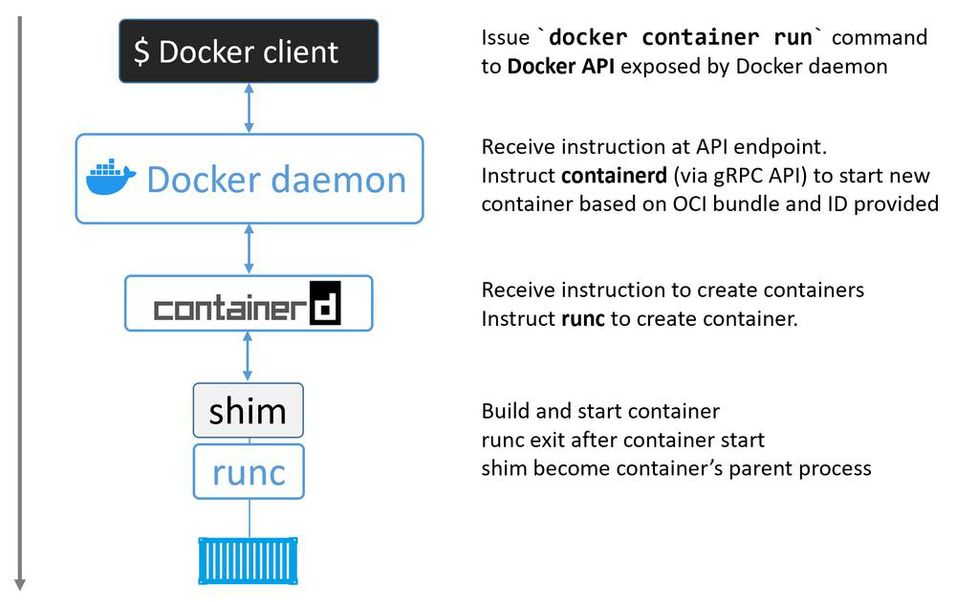
A diagram of a container engine

Description automatically generated

**High-level view of the current Docker engine architecture with brief descriptions?**



**What happens when we run command “docker container run”?**



**Can you explain docker image in simpler terms?**

A Docker image is a unit of packaging that contains everything required for an application to run. This includes; application code, application dependencies, and OS constructs. If you have an application’s Docker image, the only other thing you need to run that application is a computer running Docker.

You get Docker images by pulling them from an image registry. The most common registry is [Docker Hub](https://hub.docker.com/), but others exist. The pull operation downloads the image to your local Docker host where Docker can use it to start one or more containers.

**Images** are like stopped containers (or **classes** if you’re a developer). In fact, you can stop a container and create a new image from it. With this in mind**, images are considered build-time constructs**, whereas **containers are run-time constructs**.

**Can you give high level relationships between images and containers?**

We use the docker container run and docker service create commands to start one or more containers from a single image. Once you’ve started a container from an image, the two constructs become dependent on each other and you cannot delete the image until the last container using it has been stopped and destroyed. Attempting to delete an image without stopping and destroying all containers using it will result in an error.

A long black line on a white background

Description automatically generated

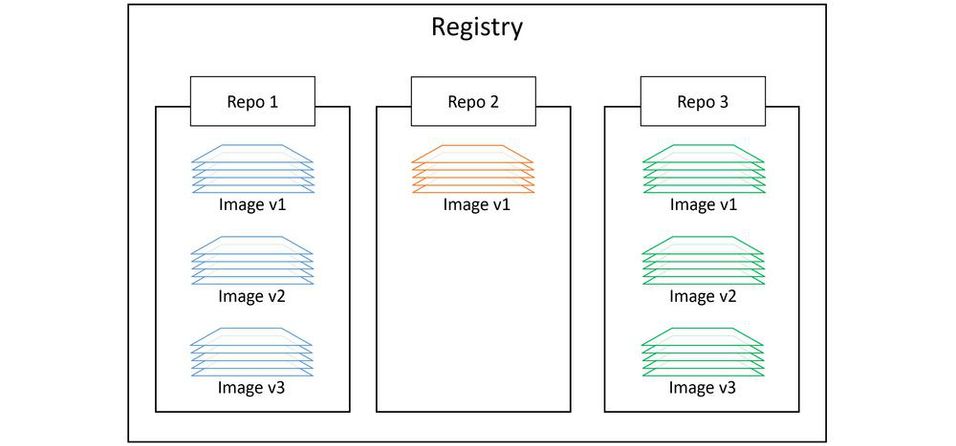
**Can you explain docker image registry?**

We store images in centralised places called image registries. This makes it easy to share and access them.

The most common registry is Docker Hub ([https://hub.docker.com](https://hub.docker.com/)). Other registries exist, including 3rd party registries and secure on-premises registries. However, the Docker client is opinionated and defaults to using Docker Hub. We’ll be using Docker Hub for the rest of the book.

The output of the following command is snipped, but you can see that Docker is configured to use <https://index.docker.io/v1/> as its default registry when pushing and pulling images (this actually redirects to v2).

Image registries contain one or more image repositories. In turn, image repositories contain one or more images. That might be a bit confusing, so [Figure 6.2](https://cdn2.percipio.com/1691311915.af10dca184ae3cb15b3d6d00505fb3fea2a147ce/eod/books/155691/OEBPS/section-38.xhtml#ch06fig2) shows a picture of an image registry with 3 repositories, and each repository has one or more images.



**What is significance of choosing between Official and Unofficial Repositories?**

Docker Hub has the concept of official repositories and unofficial repositories.

As the name suggests, official repositories are the home to images that have been vetted and curated by Docker, Inc. This means they should contain up-to-date, high-quality code, that is secure, well-documented, and in-line with best practices.

Unofficial repositories can be like the wild-west — you should not assume they are safe, well-documented or built according to best practices. That’s not saying everything in unofficial repositories is bad. There’s some excellent stuff in unofficial repositories. You just need to be very careful before trusting code from them. To be honest, you should always be careful when trusting software from the internet — even images from official repositories.

Most of the popular applications and base operating systems have their own official repositories on Docker Hub. They’re easy to spot because they live at the top level of the Docker Hub namespace. The following list contains a few of the official repositories, and shows their URLs that exist at the top-level of the Docker Hub namespace:

**What is usage of Image Naming and Tagging?**

Addressing images from official repositories is as simple as providing the repository name and tag separated by a colon (:). The format for docker image pull, when working with an image from an official repository is:

$ docker image pull <repository>:<tag>

First, if you **do not** specify an image tag after the repository name, Docker will assume you are referring to the image tagged as latest. If the repository doesn’t have an image tagged as latest the command will fail.

Second, the latest tag doesn’t have any magical powers. Just because an image is tagged as latest does not guarantee it is the most recent image in a repository. For example, the most recent image in the alpine repository is usually tagged as edge. Moral of the story — take care when using the latest tag.

Pulling images from an *unofficial repository* is essentially the same — you just need to prepend the repository name with a Docker Hub username or organization name. The following example shows how to pull the v2 image from the tu-demo repository owned by a not-to-be-trusted person whose Docker Hub account name is nigelpoulton.

$ docker image pull nigelpoulton/tu-demo:v2

//This will pull the image tagged as 'v2'

//from the 'tu-demo' repository within the 'nigelpoulton' namespace

If you want to pull images from 3rd party registries (not Docker Hub), you need to prepend the repository name with the DNS name of the registry. For example, the following command pulls the 3.1.5 image from the google-containers/git-sync repo on the Google Container Registry (gcr.io).

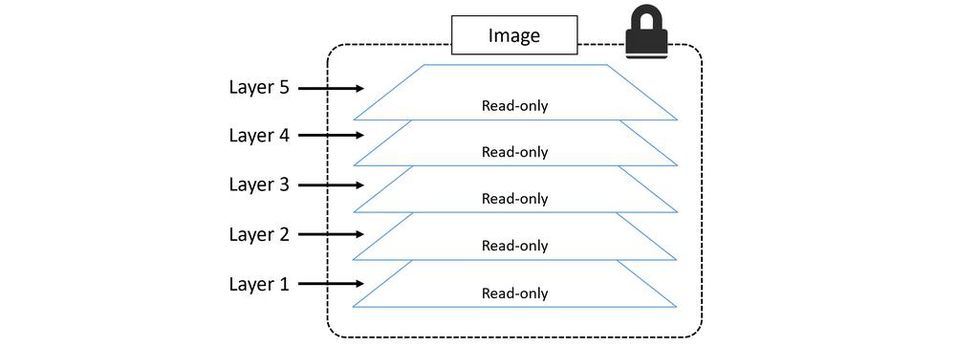
**What are dangling images?**

A dangling image is an image that is no longer tagged, and appears in listings as <none>:<none>. A common way they occur is when building a new image giving it a tag that already exists. When this happens, Docker will build the new image, notice that an existing image already has the same tag, remove the tag from the existing image and give it to the new image.

We can delete all dangling images on a system with the docker image prune command. If you add the -a flag, Docker will also remove all unused images (those not in use by any containers).

**Can you explain images and its layers?**

A Docker image is just a bunch of loosely-connected read-only layers, with each layer comprising one or more files.

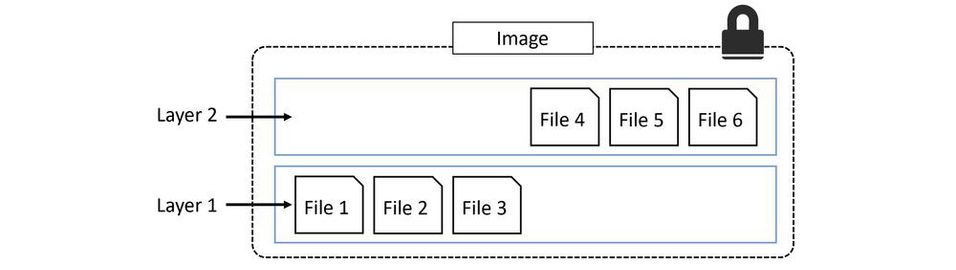


Consider the following oversimplified example of building a simple Python application. You might have a corporate policy that all applications are based on the official Ubuntu 20:04 image. This would be your image’s base layer. If you then add the Python package, this will be added as a second layer on top of the base layer. If you later add source code files, these will be added as additional layers. Your final image would have three layers as shown in below figure (remember this is an over-simplified example for demonstration purposes).

A diagram of a software development process

Description automatically generated

Each layer has 3 files, but the overall image has 6 files as it is the combination of both layers.



the overall image only presents 6 files in the unified view. This is because File 7 in the top layer is an updated version of File 5 directly below (inline). In this situation, the file in the higher layer obscures the file directly below it. This allows updated versions of files to be added as new layers to the image.

A screenshot of a computer

Description automatically generated

Docker employs a storage driver that is responsible for stacking layers and presenting them as a single unified filesystem/image.

A screenshot of a computer

Description automatically generated

**Can you explain what issues can occur while deleting images?**

When you no longer need an image on your Docker host, you can delete it with the docker image rm command. rm is short for remove.

Deleting an image will remove the image and all of its layers from your Docker host. This means it will no longer show up in docker image ls commands and all directories on the Docker host containing the layer data will be deleted. However, if an image layer is shared by more than one image, that layer will not be deleted until all images that reference it have been deleted.

Delete the images pulled in the previous steps with the docker image rm command. The following example deletes an image by its ID, this might be different on your system.

$ docker image rm 02674b9cb179

Untagged: alpine@sha256:c0537ff6a5218...c0a7726c88e2bb7584dc96

Deleted: sha256:02674b9cb179d57...31ba0abff0c2bf5ceca5bad72cd9

Deleted: sha256:e154057080f4063...2a0d13823bab1be5b86926c6f860

You can list multiple images on the same command by separating them with whitespace like the following.

$ docker image rm f70734b6a266 a4d3716dbb72

If the image you are trying to delete is in use by a running container you will not be able to delete it. Stop and delete any containers before trying the delete operation again.

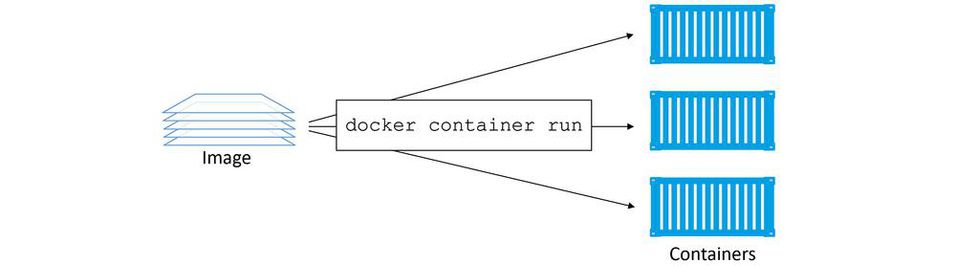
A handy shortcut for **deleting all images** on a Docker host is to run the docker image rm command and pass it a list of all image IDs on the system by calling docker image ls with the -q flag. This is shown next.

**What is a docker container?**

A container is the runtime instance of an image.

A container is the runtime instance of an image. In the same way that you can start a virtual machine (VM) from a virtual machine template, you start one or more containers from a single image. The big difference between a VM and a container is that containers are faster and more lightweight — instead of running a full-blown OS like a VM, containers share the OS/kernel with the host they’re running on. It’s also common for containers to be based on minimalist images that only include software and dependencies required by the application.

Below figure single Docker image being used to start multiple Docker containers.



The simplest way to start a container is with the docker container run command. The command can take a lot of arguments, but in its most basic form you tell it an image to use and a app to run: docker container run <image> <app>. The following command will start an Ubuntu Linux container running the Bash shell as its app.$ docker container run -it ubuntu /bin/bash

You can use the following command to start a Windows container running the PowerShell app.

> docker container run -it mcr.microsoft.com/powershell:nanoserver pwsh.exe

**How to start and stop a container ?**

You can manually stop a running container with the docker container stop command. You can then restart it with docker container start. To get rid of a container forever, you have to explicitly delete it with docker container rm.

**What is hypervisor?**

A hypervisor is a specialised OS that is highly tuned for VMs.

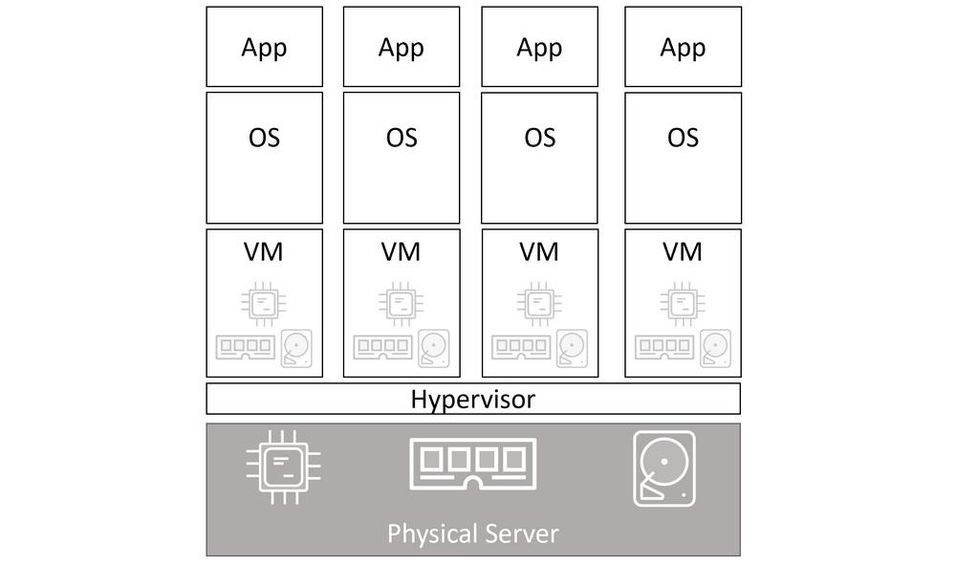
**Can you explain differences between Containers vs VMs?**

Containers and VMs both need a host to run on. This can be anything from your laptop, a bare metal server in your data center, all the way up to an instance in the public cloud. In fact, many cloud services now offer the ability to run containers on ephemeral serverless back-ends. Don’t worry if that sounds like techno-babble, it just means that the back-end is so highly virtualized that the concept of a host or node no longer has any meaning — your container simply runs, and you don’t need to care about the how or where.

Anyway… let’s assume a requirement where your business has a single physical server that needs to run 4 business applications.

In the VM model, the physical server is powered on and the hypervisor boots (we’re skipping the BIOS and bootloader code etc.). Once booted, the hypervisor lays claim to all physical resources on the system such as CPU, RAM, storage, and NICs. It then carves these hardware resources into virtual versions that look smell and feel exactly like the real thing. It then packages them into a software construct called a virtual machine (VM). We take those VMs and install an operating system and application on each one.

Assuming the scenario of a single physical server that needs to run 4 business applications, we’d create 4 VMs, install 4 operating systems, and then install the 4 applications. When it’s all done it looks a bit like below figure.



If we assume the same scenario of a single physical server needing to run 4 business applications, we’d carve the OS into 4 containers and run a single application inside each.

A diagram of a device

Description automatically generated

At a high level, hypervisors perform **hardware virtualization** — they carve up physical hardware resources into virtual versions called VMs. On the other hand, containers perform **OS virtualization** — they carve OS resources into virtual versions called containers.

**What is VM Tax?**

We started out with a single physical server and the requirement to run 4 business applications. In both models we installed either an OS or a hypervisor (a specialised OS that is highly tuned for VMs). So far, the models are almost identical. But this is where the similarities stop.

The VM model carves **low-level hardware resources** into VMs. Each VM is a software construct containing virtual CPUs, virtual RAM, virtual disks etc. As such, every VM needs its own OS to claim, initialize, and manage all of those virtual resources. And sadly, every OS comes with its own set of baggage and overheads. For example, every OS consumes a slice of CPU, a slice of RAM, a slice of storage etc. Some need their own licenses, as well as people and infrastructure to patch and upgrade them. Each OS also presents a sizable attack surface. We often refer to all of this as the **OS tax**, or **VM tax** — every OS you install consumes resources!

The container model has a single OS/kernel running on the host. It’s possible to run tens or hundreds of containers on a single host with every container sharing that single OS/kernel. That means a single OS consuming CPU, RAM, and storage. A single OS that needs licensing. A single OS that needs updating and patching. And a single OS kernel presenting an attack surface. All in all, a single OS tax bill!

That might not seem a lot in our example of a single server running 4 business applications. But when you start talking about hundreds or thousands of apps, it becomes a game-changer.

Another thing to consider is application start times. As a container isn’t a full-blown OS, it starts **much faster** than a VM. Remember, there’s no kernel inside of a container that needs locating, decompressing, and initializing — not to mention all of the hardware enumerating and initializing associated with a normal kernel bootstrap. None of that is needed when starting a container. The single shared kernel, running on the host machine, is already started. Net result, containers can start in less than a second. The only thing that has an impact on container start time is the time it takes to start the application it’s running.

This all amounts to the container model being leaner and more efficient than the VM model. You can pack more applications onto less resources, start them faster, and pay less in licensing and admin costs, as well as present less of an attack surface to the dark side.

Well, one thing that’s not so great about the container model is security. Out of the box, containers are less secure and provide less workload isolation than VMs. Technologies exist to secure containers and lock them down, but at the time of writing, some of them are prohibitively complex.

**What will happen when you run below command**

**docker *container run -it ubuntu:latest /bin/bash*?**

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

d51af753c3d3: Pull complete

fc878cd0a91c: Pull complete

6154df8ff988: Pull complete

fee5db0ff82f: Pull complete

Digest: sha256:747d2dbbaaee995098c9792d99bd333c6783ce56150d1b11e333bbceed5c54d7

Status: Downloaded newer image for ubuntu:latest

root@50949b614477:/#

When you hit Return, the Docker client packaged up the command and POSTed it to the API server running on the Docker daemon. The Docker daemon accepted the command and searched the Docker host’s local image repository to see if it already had a copy of the requested image. In the examples cited, it didn’t, so it went to Docker Hub to see if it could find it there. It found it, pulled it locally, and stored it in its local cache.

Once the image was pulled, the daemon instructed containerd and runc to create and start the container.

If you’re following along, your terminal is now attached to the container — look closely and you’ll see that your shell prompt has changed. In the Linux example cited, the shell prompt has changed to root@50949b614477:/#. The long number after the @ is the first 12 characters of the container’s unique ID

Try executing some basic commands inside of the container. You might notice that some of them don’t work. This is because the images are optimized to be lightweight. As a result, they don’t have all of the normal commands and packages installed. The following example shows a couple of commands — one succeeds and the other one fails.

root@50949b614477:/# ls -l

total 64

lrwxrwxrwx 1 root root 7 Apr 23 11:06 bin -> usr/bin

drwxr-xr-x 2 root root 4096 Apr 15 11:09 boot

drwxr-xr-x 5 root root 360 Apr 27 17:24 dev

drwxr-xr-x 1 root root 4096 Apr 27 17:24 etc

drwxr-xr-x 2 root root 4096 Apr 15 11:09 home

lrwxrwxrwx 1 root root 7 Apr 23 11:06 lib -> usr/lib

<Snip>

root@50949b614477:/# ping nigelpoulton.com

bash: ping: command not found

As you can see, the ping utility is not included as part of the official Ubuntu image.

**What command you can use to see running processes inside a container?**

root@50949b614477:/# ps -elf

The first process in the list, with PID 1, is the Bash shell we told the container to run. The second process is the ps -elf command we ran to produce the list. This is a short-lived process that exits as soon as the output is displayed. Long story short, this container is running a single process — /bin/bash.

#### Note

*Windows containers are slightly different and tend to run quite a few background processes.*

If you’re logged on to the container and type exit, you’ll terminate the Bash process and the container will exit (terminate). This is because a container cannot exist without it’s designated main process. This is true of Linux and Windows containers — **killing the main process in the container will kill the container**.

**How to stop and delete a container?**

you should stop and delete the container with the following two commands (you will need to substitute the ID of your container).

$ docker container stop 50949b614477

50949b614477

$ docker container rm 50949b614477

50949b614477

**Why it is important to first stop container and then only delete it?**

You can delete a running container with a single command, by passing the -f flag to docker container rm. However, it’s considered a best practice to take the two-step approach of stopping the container first and then deleting it. This gives the application/process running in the container a fighting chance of stopping cleanly. More on this in a second.

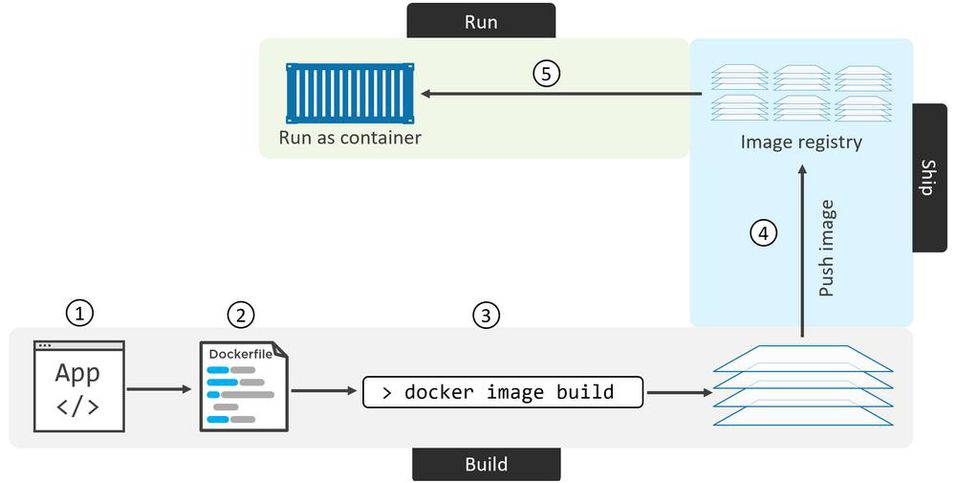
**What are steps for containerizing a web application?**

The process of containerizing an app looks like this:

1. Start with your application code and dependencies.
2. Create a Dockerfile that describes your app, its dependencies, and how to run it
3. Feed the Dockerfile into the docker image build command
4. Push the new image to a registry (optional)
5. Run container from the image

Once your app is containerized (made into a container image), you’re ready to share it and run it as a container.

Basic flow of containerizing an app



**What is purpose moving to Production with Multi-Stage Builds?**

The Dockerfile is shown below:

FROM node:latest AS storefront

WORKDIR /usr/src/atsea/app/react-app

COPY react-app .

RUN npm install

RUN npm run build

FROM maven:latest AS appserver

WORKDIR /usr/src/atsea

COPY pom.xml .

RUN mvn -B -f pom.xml -s /usr/share/maven/ref/settings-docker.xml dependency:resolve

COPY . .

RUN mvn -B -s /usr/share/maven/ref/settings-docker.xml package -DskipTests

FROM java:8-jdk-alpine AS production

RUN adduser -Dh /home/gordon gordon

WORKDIR /static

COPY --from=storefront /usr/src/atsea/app/react-app/build/ .

WORKDIR /app

COPY --from=appserver /usr/src/atsea/target/AtSea-0.0.1-SNAPSHOT.jar .

ENTRYPOINT ["java", "-jar", "/app/AtSea-0.0.1-SNAPSHOT.jar"]

CMD ["--spring.profiles.active=postgres"]

$ git clone <https://github.com/nigelpoulton/atsea-sample-shop-app.git>

Cloning into 'atsea-sample-shop-app'...

remote: Counting objects: 632, done.

remote: Total 632 (delta 0), reused 0 (delta 0), pack-reused 632

Receiving objects: 100% (632/632), 7.23 MiB | 1.88 MiB/s, done.

Resolving deltas: 100% (195/195), done.

Checking connectivity... done.

$ cd atsea-sample-shop-app/app

$ ls -l

total 24

-rw-r--r-- 1 root root 682 Oct 1 22:03 Dockerfile

-rw-r--r-- 1 root root 4365 Oct 1 22:03 pom.xml

drwxr-xr-x 4 root root 4096 Oct 1 22:03 react-app

drwxr-xr-x 4 root root 4096 Oct 1 22:03 src

$ docker image build -t multi:stage .

Sending build context to Docker daemon 3.658MB

Step 1/19 : FROM node:latest AS storefront

latest: Pulling from library/node

aa18ad1a0d33: Pull complete

15a33158a136: Pull complete

<Snip>

Step 19/19 : CMD --spring.profiles.active=postgres

---> Running in b4df9850f7ed

---> 3dc0d5e6223e

Removing intermediate container b4df9850f7ed

Successfully built 3dc0d5e6223e

Successfully tagged multi:stage

$ docker image ls

REPO TAG IMAGE ID CREATED SIZE

node latest a5a6a9c32877 5 days ago 941MB

<none> <none> d2ab20c11203 9 mins ago 1.11GB

maven latest 45d27d110099 9 days ago 508MB

<none> <none> fa26694f57cb 7 mins ago 649MB

java 8-jdk-alpine 3fd9dd82815c 7 months ago 145MB

multi stage 3dc0d5e6223e 1 min ago 210MB

The top line in the output above shows the node:latest image pulled by the storefront stage. The image below is the image produced by that stage (created by adding the code and running the npm install and build operations). Both are very large images with lots of build junk included.

The 3rd and 4th lines are the images pulled and produced by the appserver stage. These are both large and contain lots of builds tools.

The last line is the multi:stage image built by the final build stage in the Dockerfile (stage2/production). You can see that this is significantly smaller than the images pulled and produced by the previous stages. This is because it’s based off the much smaller java:8-jdk-alpine image and has only added the production-related app files from the previous stages.

The net result is a small production image created by a single Dockerfile, a normal docker image build command, and zero additional scripting!

Multi-stage builds were new with Docker 17.05 and are an excellent feature for building small production-worthy images.

**What are some of the best practices related to building a docker image?**

**Leverage the Build Cache**

Firstly, as soon as any instruction results in a cache-miss (no layer was found for that instruction), the cache is no longer used for the rest of the entire build. This has an important impact on how you write your Dockerfiles. Try and write them in a way that places instructions that are likely to invalidate the cache towards the end of the Dockerfile. This means that a cache-miss will not occur until later stages of the build - allowing the build to benefit as much as possible from the cache.

You can force the build process to ignore the entire cache by passing the --no-cache=true flag to the docker image build command.

It is also important to understand that the COPY and ADD instructions include steps to ensure that the content being copied into the image has not changed since the last build. For example, it’s possible that the COPY . /src **instruction** in the Dockerfile has not changed since the previous, **but…** the contents of the directory being copied into the image **have** changed!

To protect against this, Docker performs a checksum against each file being copied, and compares that to a checksum of the same file in the cached layer. If the checksums do not match, the cache is invalidated and a new layer is built.

**Squash the Image:**

Squashing an image isn’t really a best practice as it has pros and cons.

At a high level, Docker follows the normal process to build an image, but then adds an additional step that squashes everything into a single layer.

Squashing can be good in situations where images are starting to have a lot of layers and this isn’t ideal. An example might be when creating a new base image that you want to build other images from in the future — this base is much better as a single-layer image.

On the negative side, squashed images do not share image layers. This can result in storage inefficiencies and larger push and pull operations.

A screenshot of a computer program

Description automatically generated

**Use No-Install-Recommends:**

If you are building Linux images, and using the apt package manager, you should use the no-install-recommends flag with the apt-get install command. This makes sure that apt only installs main dependencies (packages in the Depends field) and not recommended or suggested packages. This can greatly reduce the number of unwanted packages that are downloaded into your images.

**What are different commands related to containerizing an app?**

* docker image build is the command that reads a Dockerfile and containerizes an application. The -t flag tags the image, and the -f flag lets you specify the name and location of the Dockerfile. With the -f flag, it is possible to use a Dockerfile with an arbitrary name and in an arbitrary location. The build context is where your application files exist, and this can be a directory on your local Docker host or a remote Git repo.
* The FROM instruction in a Dockerfile specifies the base image for the new image you will build. It is usually the first instruction in a Dockerfile and a best-practice is to use images from official repos on this line.
* The RUN instruction in a Dockerfile allows you to run commands inside the image. Each RUN instruction creates a single new layer.
* The COPY instruction in a Dockerfile adds files into the image as a new layer. It is common to use the COPY instruction to copy your application code into an image.
* The EXPOSE instruction in a Dockerfile documents the network port that the application uses.
* The ENTRYPOINT instruction in a Dockerfile sets the default application to run when the image is started as a container.
* Other Dockerfile instructions include LABEL, ENV, ONBUILD, HEALTHCHECK, CMD and more…

**What is use of docker compose?**

Modern cloud-native apps are made of multiple smaller services that interact to form a useful app. We call this pattern “microservices”. A simple example might be an app with the following seven services:

* Web front-end
* Ordering
* Catalog
* Back-end database
* Logging
* Authentication
* Authorization

Get all of these working together, and you have a useful application.

Deploying and managing lots of small microservices like these can be hard. This is where Docker Compose comes in to play.

Instead of gluing each microservice together with scripts and long docker commands, Docker Compose lets you describe an entire app in a single declarative configuration file, and deploy it with a single command.

Once the app is deployed, you can manage its entire lifecycle with a simple set of commands. You can even store and manage the configuration file in a version control system.

**What are some common docker compose commands?**

* docker-compose up is the command we use to deploy a Compose app. It expects the Compose file to be called docker-compose.yml or docker-compose.yaml, but you can specify a custom filename with the -f flag. It's common to start the app in the background with the -d flag.
* docker-compose stop will stop all of the containers in a Compose app without deleting them from the system. The app can be easily restarted with docker-compose restart.
* docker-compose rm will delete a stopped Compose app. It will delete containers and networks, but it will not delete volumes and images.
* docker-compose restart will restart a Compose app that has been stopped with docker-compose stop. If you have made changes to your Compose app since stopping it, these changes will **not** appear in the restarted app. You will need to re-deploy the app to get the changes.
* docker-compose ps will list each container in the Compose app. It shows current state, the command each one is running, and network ports.
* docker-compose down will stop and delete a running Compose app. It deletes containers and networks, but not volumes and images.

**What are rules for method overriding?**

Rules for method overriding:

1.The argument list should be exactly the same as that of the overridden method.

2.The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

3.The access level cannot be more restrictive than the overridden method’s access level. For example: if the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.

4.Instance methods can be overridden only if they are inherited by the subclass.

5.A method declared as final cannot be overridden.

6.A method declared as static cannot be overridden but can be re-declared.

7.If a method cannot be inherited, then it cannot be overridden.

8.A subclass within the same package as the instance’s superclass can override any superclass method that is not declared private or final.

9.A subclass in a different package can only override the non-final methods declared public or protected.

10.An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not.

11.However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method.

12.The overriding method can throw narrower or fewer exceptions than the overridden method.

13.Constructors cannot be overridden.

**What is abstract class in Java?**

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

In other words it is the process to show the features which are necessary for current problem scenario.

For example, if you want to switch on the fan, you will click on the switch but you don’t know the internal processing about the fan.

There are two ways to achieve abstraction in Java

1.Abstract class (0 to 100%)

2.Interface (100%)

Abstract Class

1.A class which contains implemented and non implemented methods, declared as abstract and called abstract class.

2.Abstract class cannot be instantiated.

3.Abstract classes may or may not contain abstract methods, i.e., methods without body ( public void fun(); )

4.To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.

5.If you inherit an abstract class, you have to provide implementations to all the abstract methods in it otherwise derived class should be declare as abstract.

**What is heap dump and can you explain with a sample example?**

A heap dump is like a map of the heap memory. If you learn how to read it, it gives you invaluable clues about how the app internally processes data. A heap dump helps you investigate memory problems or performance issues.

public class Main {

private static List<Product> products = ❶

new ArrayList<>();

public static void main(String[] args) {

while (true) {

products.add( ❷

new Product(UUID.randomUUID().toString())); ❸

}

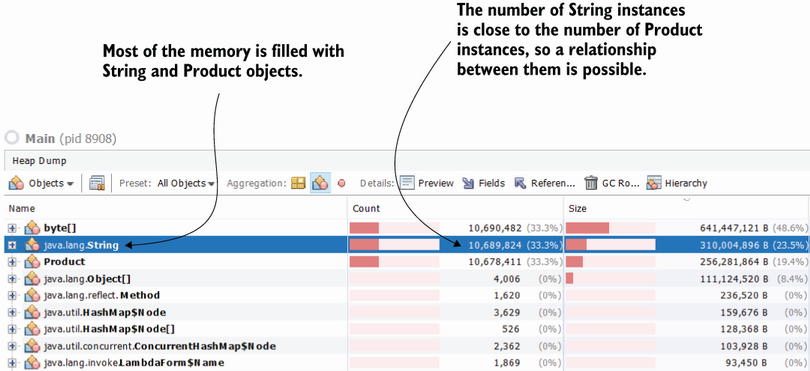
}

}

In this example, you can easily find which object fills most of the app's memory and that the Product and String instances are related.

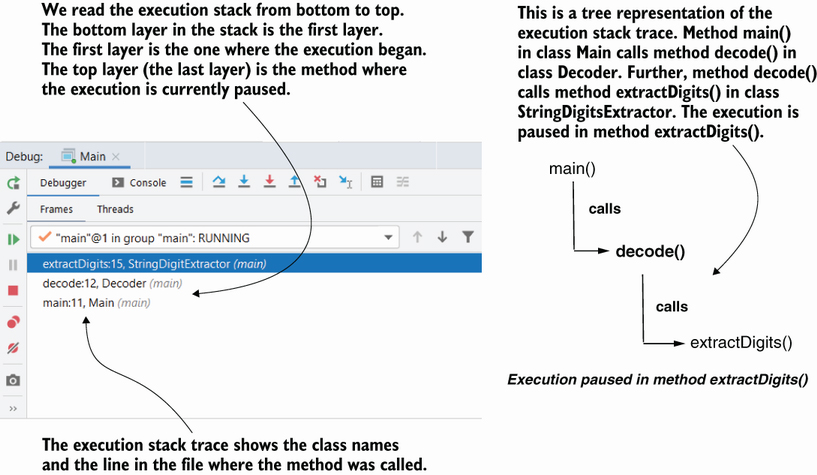
A thread dump provides details about the threads that were running when the dump was taken. It includes thread states and the stack traces, which tell you what the threads were executing or what blocked them. These details are valuable for investigating why an app is stuck or is having performance problems





**What is the Execution Stack Trace, and How do I Use It?**

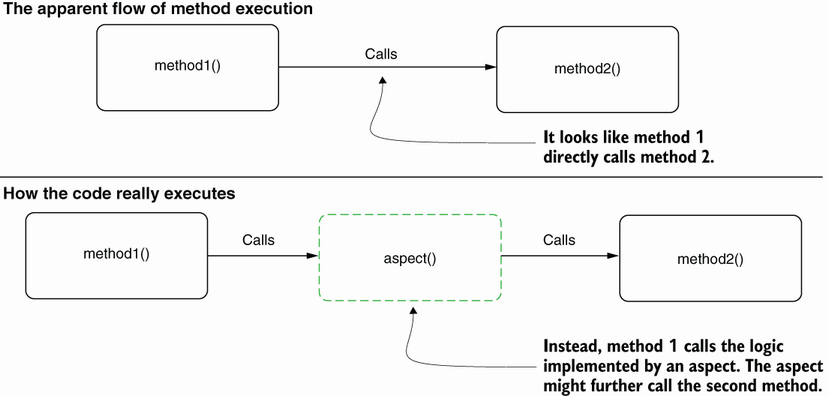
The execution stack trace is a valuable tool you use to understand the code while debugging it. Just like a map, the execution stack trace shows you the execution's path to the specific line of code where the debugger paused it and helps you to decide where to navigate further.



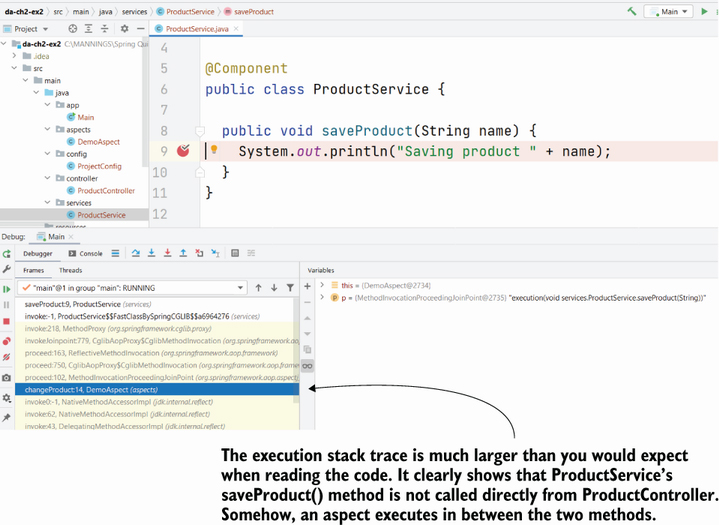
**Can you explain how aspects in Spring framework?**

One of my favorite uses of the execution stack trace is finding hidden logic in the execution path. In most cases, developers use the execution stack trace simply to understand where a certain method has been called from. But you also need to consider that apps that use frameworks (such as Spring, Hibernate, etc.) sometimes alter the execution chain of the method.

For example, Spring apps often use code that is decoupled in what is called aspects (in Java/Jakarta EE terminology, they are named interceptors). These aspects implement logic that the framework uses to augment the execution of specific methods in certain conditions. Unfortunately, such logic is often difficult to observe since you can't see the aspect code directly in the call chain when reading the code (figure 2.9). This characteristic makes it challenging to investigate a given capability.



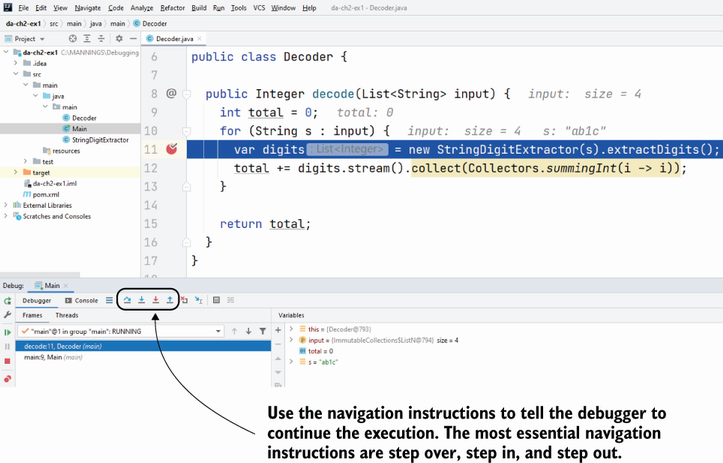
The execution stack trace shows that an aspect has altered the execution. This aspect is the reason that the value of the parameter changes. Without using the stack trace, finding why the app has a different behavior than expected would be more difficult



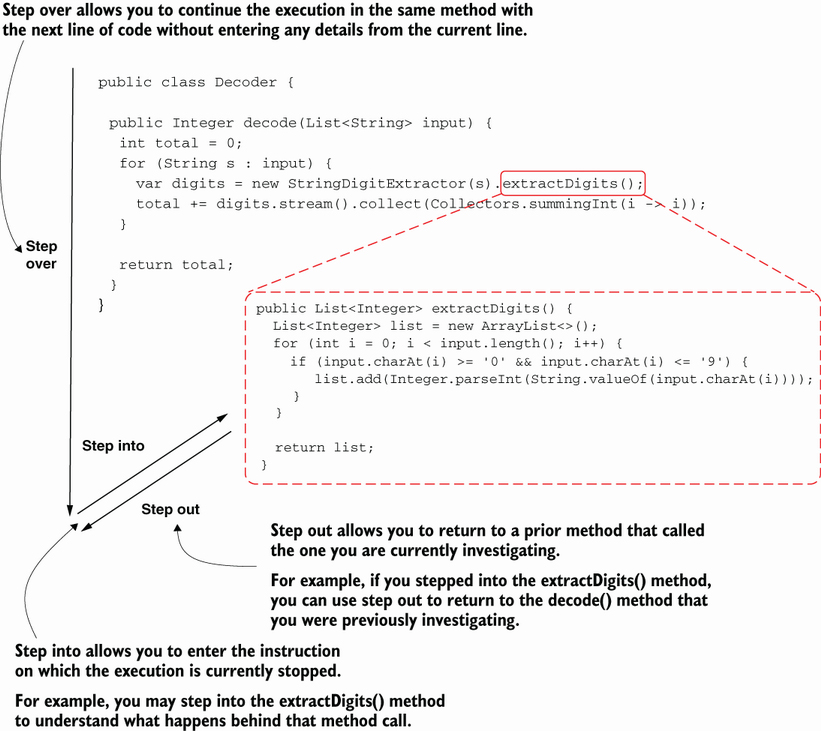
**What are navigation options while debugging?**

* Step over—Continue the execution with the next line of code in the same method.
* Step into—Continue the execution inside one of the methods called on the current line.
* Step out—Return the execution to the method that called the one you are investigating.

Figure 2.12: The navigation operations help you “walk” through the app logic in a controlled way to identify how the code works. To navigate through code, you can use the buttons on the IDE's GUI or use the keyboard shortcuts associated with these operations

* 

Navigation operations. Stepping over allows you to go to the next instruction in the same method. When you want to start a new investigation plan and go into detail in a specific instruction, you can use the step into operation. You can go back to the previous investigation plan with the step out operation.



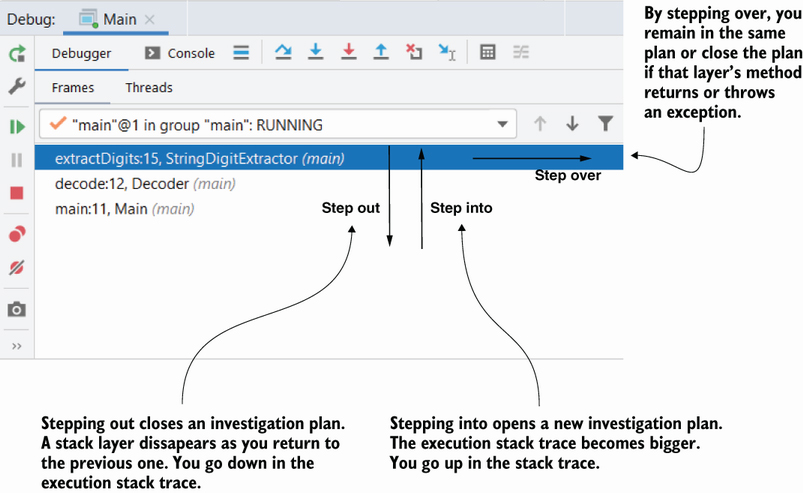
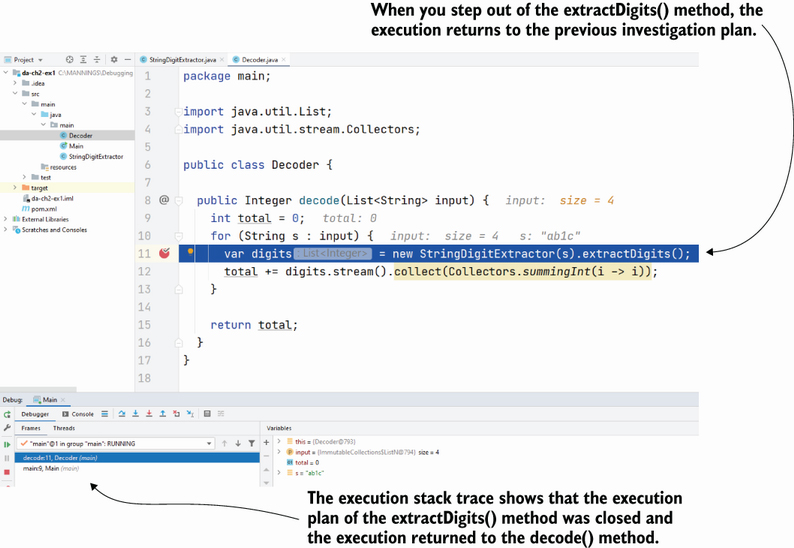
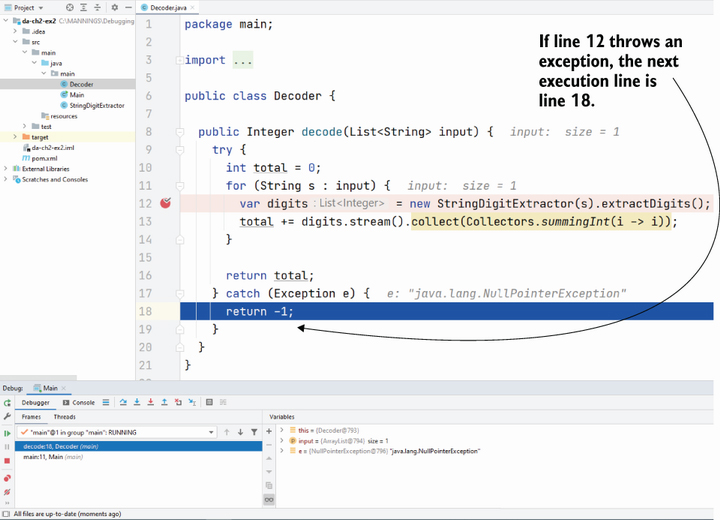


Figure 2.18: The step out operation allows you to close an investigation plan and return to the previous one in the execution stack trace. Using step out is helpful to save time since you don't have to step over each instruction until the current execution plan closes by itself. Stepping out offers you a shortcut to return to the previous execution plan you were investigating



**Why is the next execution line not always the next line?**

When using the step over operation, the execution will continue to the *next execution line*.



**When using the Debugger Might Not Be Enough?**

The debugger is an excellent tool that can help you to analyze code by navigating through the code to understand how it works with data. But not all code can be investigated with a debugger. In this section, we discuss some scenarios in which using a debugger is not possible or not enough. You need to be aware of these cases so that you don't waste time using a debugger.

Here are some of the most often encountered investigation scenarios when using a debugger (or only a debugger) is usually not the right approach:

* Investigating output problems when you don't know which part of the code creates the output
* Investigating performance problems
* Investigating crashes where the entire app fails
* Investigating multithreaded implementations

**Why it is hard to debug multithreaded application code?**

Most developers find multithreaded implementations the most challenging to investigate. Such implementations can be easily influenced by your interference with tools such as a debugger. This interference creates a Heisenberg effect (discussed in chapter 1): the app behaves differently when you use the debugger than when you don't interfere with it. As you'll learn, you can sometimes isolate the investigation to one thread and use the debugger. But in most cases, you'll have to apply a set of techniques that include debugging, mocking and stubbing, and profiling to understand the app's behavior in the most complex scenarios.

**What are some advanced debugging techniques?**

1. Using conditional breakpoints to investigate specific scenarios

2. Using breakpoints to log debug messages in the console

3. Changing data while debugging to force the app to act in a specific way

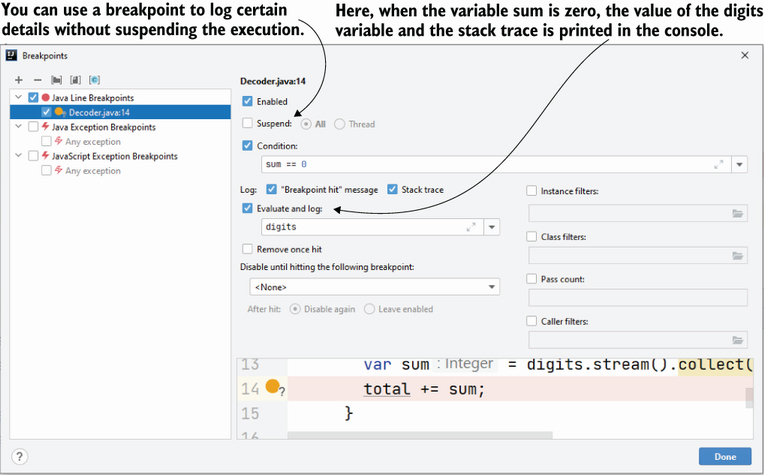
4. Rerunning a certain part of the code while debugging.

**What is conditional breakpoint?**

A conditional breakpoint is a breakpoint you associate with a condition, so that the debugger only pauses the execution if the condition is fulfilled. Conditional breakpoints are helpful in investigation scenarios when you are only interested in how a part of the code works with given values; using conditional breakpoints where appropriate saves you time and helps you to more easily understand how your app works.

**Is there any way to print variable name using conditional break in Intellij?**

Conditional breakpoint advanced configuration. In addition to specifying a condition for the breakpoint, you can instruct the debugger to not suspend the execution for the given breakpoint. Instead, you can simply log the data you need to understand your case



**How can you alter investigation scenario by altering variable value?**

 Setting a new value in a variable in scope. The debugger shows you the values for the variables in scope when it pauses the execution on a given line. You can also change the values to create a new investigation case. In some cases, this approach can help you to validate your suspicions about what the code does

A screenshot of a computer

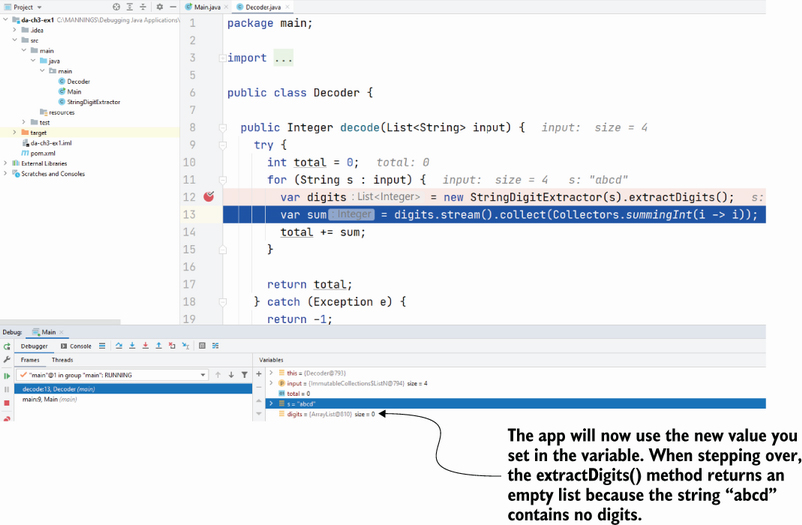
Description automatically generated

Change the variable's value to observe how the app's execution behaves in different conditions

A screenshot of a computer

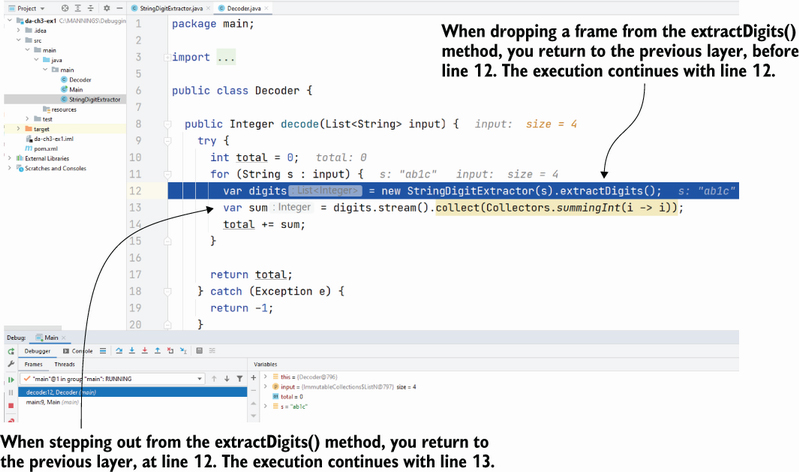
Description automatically generated

 When using the step over operation, the app uses the new value you set to the s variable. extractDigits() returns an empty list because string “abcd” doesn't contain digits. Setting values in variables on the fly allows you to test different scenarios even if you don't have the input data you need.



**How can drop frame to back to previous line of debugger?**

Dropping a frame versus stepping out. When you drop a frame, you return to the line before the method's execution. When you step out, you continue the execution but close the current investigation plan (represented by the current layer in the execution stack)

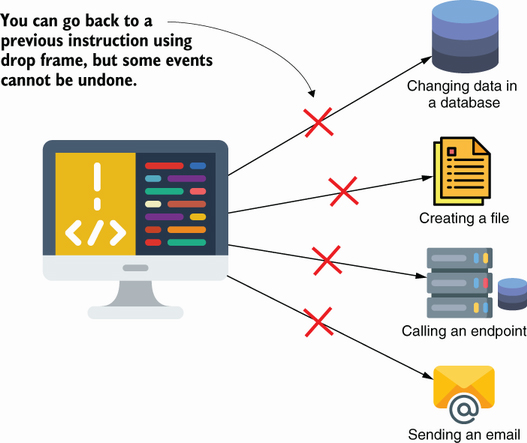


**When it is not a good idea to repeat a particular instruction by dropping frames?**

 if you run any instruction that changes values outside of the app's internal memory, you can't undo that change by dropping the frame. Examples of such cases are ([figure 3.15](https://cdn2.percipio.com/1691833178.f643599424ed3a57a426f9b7f8969a8ca38027b5/eod/books/164926/OEBPS/section-34-22.xhtml#ch03fig15)) as follows:

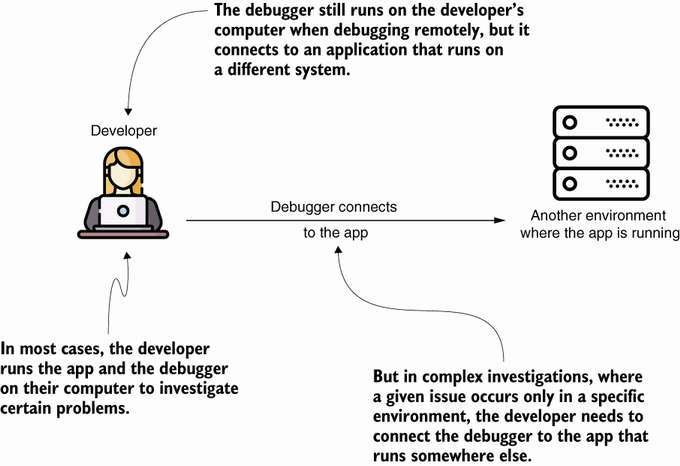
* Modifying data in a database (insert, update, or delete)
* Changing the filesystem (creating, removing, or changing files)
* Calling another app, which changes the data for that app
* Adding a message into a queue that is read by a different app, which changes data for that app
* Sending an email message

Using the drop frame operation can result in some events that can't be undone. Examples include changing data in the database, changing data in the filesystem, calling another app, or sending an email message



**How to debug applications remotely?**

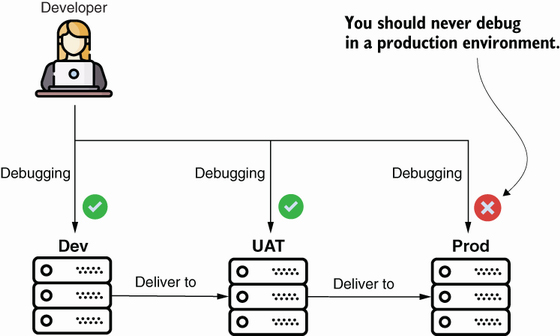
Remotely debugging an app. The developer can run the debugger tool locally but connect it to an app instance running in a different environment. This approach allows the developer to investigate problems that only occur in specific environments



**Should do remote debugging for prod environments?**

No

Developers implement the app using the dev and UAT environments. It's OK to debug apps in these environments. But remember, never debug apps in the prod environment, as this can affect the app's execution, interfere with users’ actions, and even expose sensitive data, creating a security vulnerability.

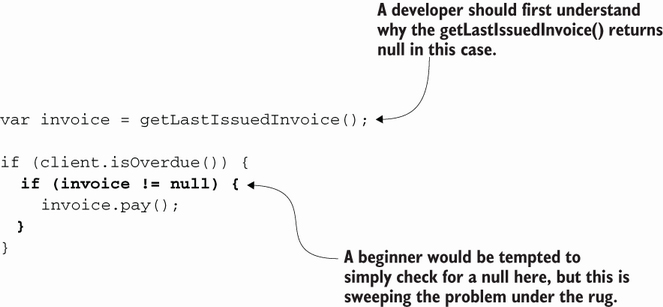


Some of the consequences of attaching the debugging agent (and why you shouldn't do this in a production environment) include these:

* The agent can slow the app's execution; this slowness can cause performance problems.
* The agent needs to communicate with the debugger tool through the network. To enable this, you need to make specific ports available, which can cause vulnerability issues.
* Debugging a specific piece of code can interfere with functionality if the same part of the app is being used elsewhere simultaneously.
* Sometimes debugging can block the app indefinitely and force you to restart the process.

**How to effectively use logs to Identify Exceptions?**

Locally solving the problem is in many cases equivalent to sweeping it under the rug. If the root cause remains, more issues can appear later. Remember that an exception in the logs doesn't necessarily indicate the root cause



**Using exception stack traces to identify what calls a method?**

#### Listing 5.1: Printing the execution stack trace in logs using an exception

public List<Integer> extractDigits() {

new Exception().printStackTrace(); ❶

List<Integer> list = new ArrayList<>();

for (int i = 0; i < input.length(); i++) {

if (input.charAt(i) >= '0' && input.charAt(i) <= '9') {

list.add(Integer.parseInt(String.valueOf(input.charAt(i))));

}

}

return list;

}

❶ Prints the exception stack trace

The next snippet shows how the app prints the exception stack trace in the console. In a real-world scenario, the stack trace helps you to immediately identify the execution flow, which leads to the call you want to investigate, as we discussed in chapters 2 and 3. In this example, you can see from the logs that the extractDigits() method was called on line 11 of the Decoder class from within the decode() method:

java.lang.Exception at main.StringDigitExtractor

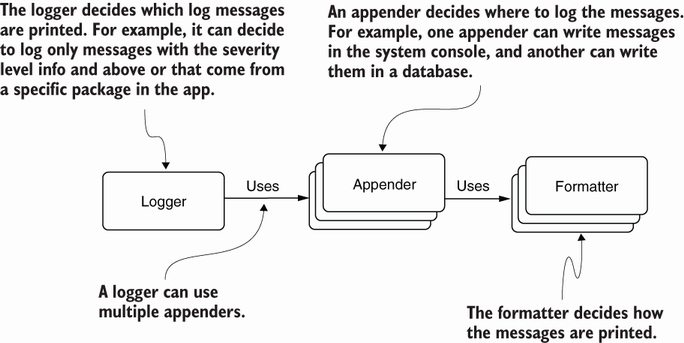
➥ extractDigits(StringDigitExtractor.java:15)

at main.Decoder.decode(Decoder.java:11)

at main.Main.main(Main.java:9)

**How do you typically log messages in Java applications?**

The relationship between the appender, logger, and formatter. A logger uses one or more appenders. The logger decides what to write (e.g., only log messages printed by objects in the package). The logger gives the messages to be written to one or more appenders. Each appender then implements a certain way to store the messages. The appender uses formatters to shape the messages before storing them



**Problems Caused by Logging and How to Avoid Them?**

* Security and privacy issues—Caused by log messages exposing private data
* Performance issues—Caused by the app storing too many or too-large log messages
* Maintainability issues—Caused by log instructions that make the source code more difficult to read

**Where Would a Profiler Be Useful?**

* 1. Identifying abnormal usage of resources
* Thread-related issues—Usually concurrency issues caused by a lack of or improper synchronization
* Memory leaks—Situations in which the app fails to remove unneeded data from memory, causing slowness in execution and potentially a complete failure of the app

2. Finding which part of code executes

3. Identifying slowness in an app's execution

**Can you explain comparison between memory usage in healthy app and app with memory leak?**

 A comparison between the memory usage for a healthy app versus an app suffering from a memory leak. The GC manages to free unneeded data from memory for a healthy app, and the allocated space never fills up. An app suffering from a memory leak doesn't allow the GC to remove enough data. At some point, the memory fills up completely, generating an OutOfMemoryError



**What is metaspace?**

The memory location where the JVM stores the class metadata needed for the app's execution.

**When to not use hibernate?**

An OutOfMemoryError on the metadata space happens less often, but it's not impossible. I recently dealt with such a case in an app that was misusing a framework for data persistence. Generally, frameworks and libraries using Java reflection are the most likely to generate such problems if misused since they often rely on dynamic proxies and indirect calls.

In my situation, the app was misusing a framework named Hibernate. I would not be surprised if you have already heard about Hibernate since it's one of the most common solutions to manage persistent data in Java apps today. Hibernate is an excellent tool that helps to implement the most-used persistence capabilities of an app while eliminating the need to write unneeded code. Hibernate manages a context of instances and maps the changes to this context to the database. But it's not recommended for a very large context. In other words, don't work with too many records from the database at once!

The app I had trouble with defined a scheduled process, loading many records from a database and processing them in a defined way. It seems that, at some point, the number of records this process was fetching was so large that the load operation itself caused the metaspace to fill; the problem was a misuse of the framework, not a bug in the framework. The developers should not have used Hibernate and instead used an alternate, more low-level solution like JDBC.

The problem was critical, and I had to find a short-term solution since a complete refactoring would have taken a long time. Just as for the heap, you can customize the metaspace size. Using the -XX:MaxMetaspaceSize property, you can enlarge the metaspace (e.g., -XX:MaxMetaspaceSize=100M), but remember that this is not a real solution to the problem. The long-term solution for such a case is to refactor the functionality to avoid loading so many records at once in the memory and eventually use an alternate persistence technology if needed.

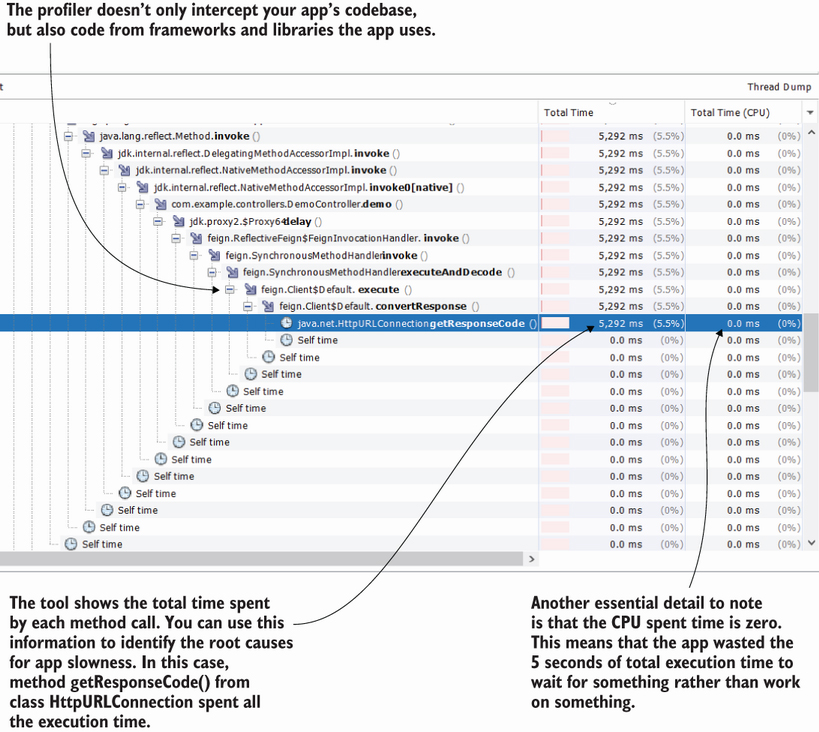
**What is purpose behind using sampling?**

Sampling the execution has three purposes:

* To find out what code executes—Sampling shows you what executes behind the scenes and is an excellent way to find the part of the app you need to investigate.
* To identify CPU consumption—We'll use this to investigate latency issues and understand which methods share execution time.
* To identify memory consumption—This allows us to analyze memory-related issues. We'll discuss sampling and profiling memory more in chapter 11.

**How do find performance issue using sampling?**

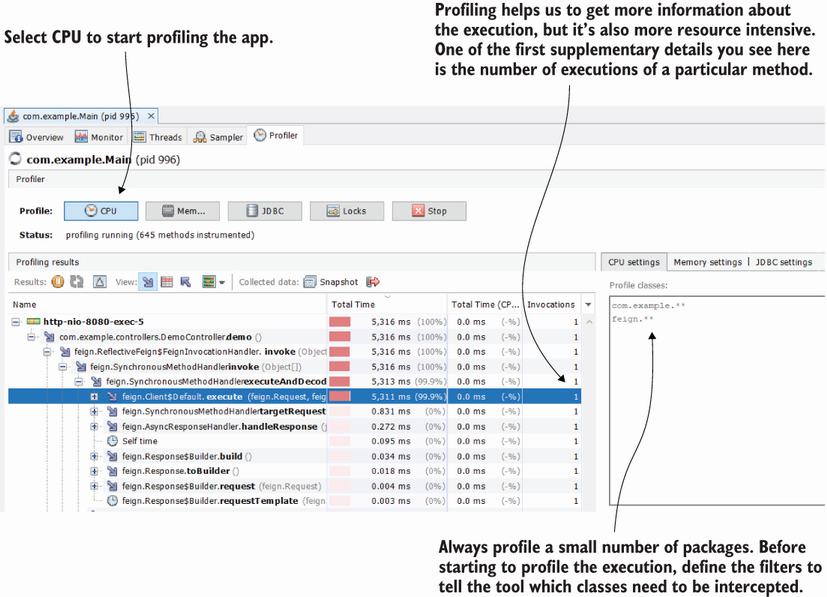
When you expand the execution stack, you find which methods execute and how much time they spend executing. You can also deduce how long they wait and how much they work. The profiler shows both the app's codebase methods and the methods called from specific dependencies (libraries or frameworks) the app uses.



*One of my personal uses for sampling is when learning a new framework or library. Sampling helps me to understand what executes behind the scenes in a new functionality. I applied this approach when learning Hibernate and Spring Security, which have complex functionality, and it helped me quickly understand how to work with the given capabilities.*

**Is there any way to profile only a part of application at package level?**

Profiling a part of the app during execution to get details about the times a given method was invoked. We can see that the method causing the 5 seconds of latency is invoked only once, meaning the number of invocations doesn't cause a problem here



The syntax you can use to filter the packages and classes you want to profile has just a few simple rules:

* Write each rule on a separate line.
* Use one asterisk (\*) to refer to a package; for example, we could use com.example.\* if we wanted to profile all classes in the package com.example.
* Use two asterisks (\*\*) to refer to a package and all its subpackages. In this case, by using com.example.\*\*, we mean all classes in the package com.example as well as any of its subpackages.
* Write the full name of a class if you want to profile only that class; for example, we could use com.example.controllers.DemoController to profile only this class.