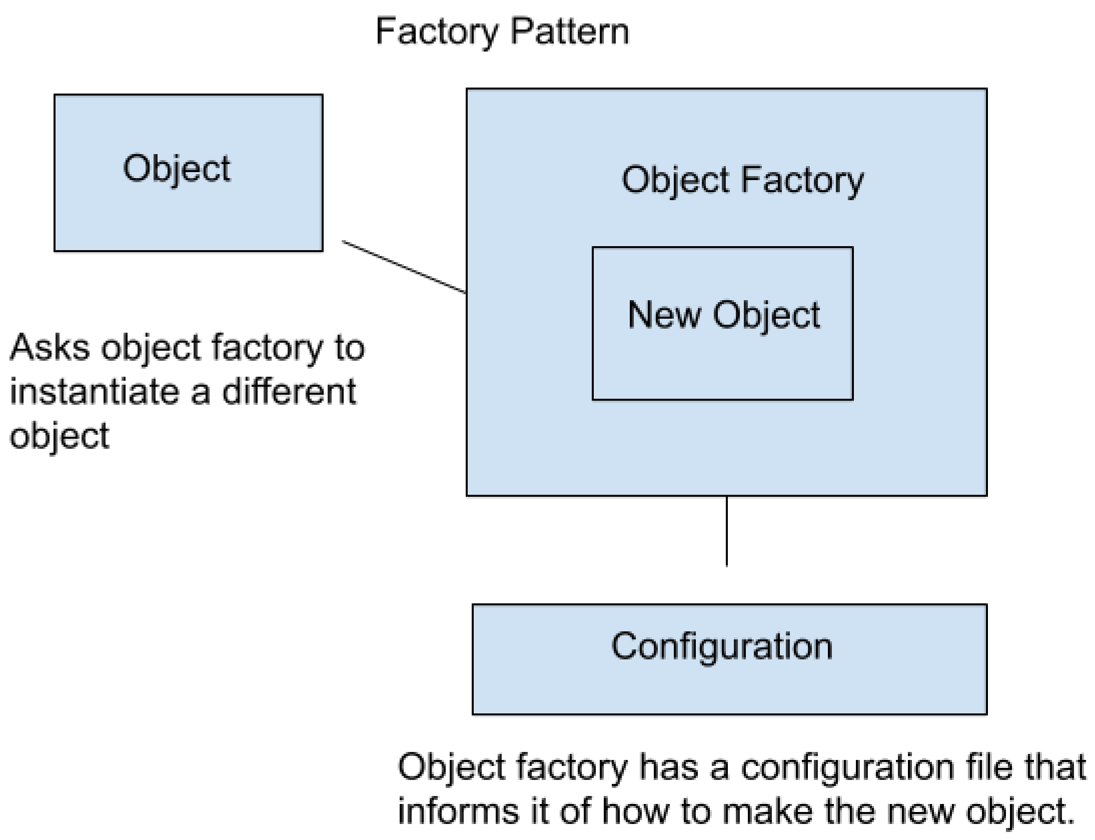
Java Stack

Alexis Abbott Notes

Table of Contents

Spring

A framework that uses AOP design principles.


Factory Pattern
Object
Object Factory
New Object
Asks object factory to instantiate a different object
Configuration
Object factory has a configuration file that informs it of how to make the new object.

Factory Bean XML lives in the root of the application in the spring.xml However, managing a Bean XML is a nightmare, so Spring moved to annotations in Boot.

```
<!DOCTYPE beans…>

<beans>
  <bean id=”objName” class=”classNamePath” />
</beans>
```

ApplicationContext does everything Bean Factory does, but has more features. They’re both interfaces.

```
ApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
```

This XML needs to go in the package folder (src).

Using a setter injector:

```
<beans>
  <bean id="objName" class="packagePath">
    <property name="type" value="nameOfField" />
  </bean>
</beans>
```

Using a constructor injector:

```
<beans>
  <bean id="objName" class="packagePath">
    <constructor-arg value="nameOfField" />
  </bean>
</beans>
```

To set more class properties, just use more <constructor-arg> tags. Since the configuration file takes in all strings, you need to tell Spring what datatype to convert to.

```
<constructor-arg type="int" value="20" />
```

You can also define an index if you want to pick which constructor you're using.

```
<constructor-arg index="0" type="int" value="20" />
```

Object Injection

```
<beans>
  <bean id="obj1" class="packagePath">
    <property name="propName" ref="obj2" />
  </bean>
  <bean id="obj2" class="packagePath">
    <property name="propName" value="someValue" />
  </bean>
</beans>
```

You can nest this as many times as needed.

Inner Beans

Place Beans that are only used by one object within the object definition.

```
<beans …>
  <bean …>
    <property …>
      <bean …>
        <property …>
        <property …>
      </bean>
    </property>
  </bean>
</beans>
```

We don’t have to give the Inner Bean an id tag. Still needs the package definition though. Within the <beans> tag, you can use an <alias> tag.

```
<alias name=”objName” alias=”my-alias” />
```

You can reference the same object with alias for different uses, when you don’t need more than one instance. We can also define an alias in the Bean definition with the attribute name.

```
<bean id=”obj1” class=”package” name=”alias1”>
```

It’s better to use IDs however since the XML verifier will error out on any duplicate IDs. If you want to restrict your object reference to an ID, use idref. It needs to be nested in the property tag.

```
<property name=”propName”>
  <idref=”obj1” />
</property>
```

If you use idref, the XML verifier will make sure it’s there or it will throw an error.

Initializing Collections

Within the <beans> tag, nested in a bean’s property tag:

```
<list>
  <ref bean=”obj1” />
  <ref bean=”obj2” />
  <ref bean=”obj3” />
</list>
```

This will be assigned to the List<ClassName> initialization in the class. You would only really want to do this with small lists.

Autowiring

Autowiring is an attribute for your config file that works like a shortcut. If your property names are the same as your bean names, you don’t have to define the properties in your bean.

```
<bean id=”obj1” class=”package” autowire=”byName”>
```

Autowire has several options. This feature however can make reading code hard. Complex applications should be explicitly wired.

Bean Factory Scope

Beans are created by default when the ApplicationContext (Bean Factory) is initialized, now when an object calls for the Bean. This can be modified in the configuration file, but this is default.

Spring Bean - An object that Spring manages instantiation.

Spring Singleton

A few differences to a standard Java Singleton. In short though, a Singleton is created only once, no matter how many getBean calls there are. Only one is instantiated in the SpringContainer(ApplicationContext).

Spring Prototype

New bean is created with every request or reference.

Singletons will be created when the container is initialized, but Prototypes are only created when they are requested or referenced.

Web-Aware Context Bean Scopes

Request - New Bean per servlet request.

Session - New Bean per session.
Global - New Bean per global HTTP session (portlet context).
To initialize the scope in the ApplicationContext XML, use the scope attribute in the Bean. The default is Singleton.

You can have your class implement the ApplicationContext interface if you need access to it. (Though this will wire your application into Spring.) By doing so, you'll have to add the interface methods to your class. The method that populates is setApplicationContext() and throws an exception.

Bean Definition Inheritance

To have Beans inherit from other Beans, use the parent attribute and name it the same as the parent bean. You can override properties in your child Beans.

There is also functionality to inherit lists. The default is to override, but if you want to combine lists from parent to child, you use the merge attribute on the <list> tag and set it to true. If you don't need the parent Bean to be initialized, you can add the abstract keyword in the XML Bean definition. This creates a "template".

Lifecycle Callbacks

Beans have a callback method that can be used to do any cleanup before the end of the lifecycle of the object. This is often more important in desktop applications as web and enterprise know when to destroy the Beans.

Initializing the container:

```
AbstractApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
context.registerShutdownHook();
```

This lets the container know that when the application ends, Spring also knows it is time to close.

InitializingBean interface used on a Bean lets Spring know the contracted method has to be called.

DisposableBean interface used on a Bean lets Spring execute the destroy() method before the Bean is destroyed. registerShutdownHook() is what fires off all the destroy methods in your application.

Avoid Wiring Spring

Essentially, create your own init() and destroy() methods without implementing Spring interfaces. In the XML config file, declare your init() and destroy() methods in the Bean definition with the attributes init-method and destroy-method.

If you use a naming convention for init and destroy, you can configure them at the global level so you don't have to define it for every single Bean. Define it in the parent <beans> tag. Use the attributes default-init-method and default-destroy-method.

Using Both

Spring's init method will be called before your custom one, and Spring's destroy will be called before your custom one. Spring is given precedence.

BeanPostProcessor

Classes used to tell Spring there's some processing that needs to be done after initalization. The most common use for this is to extend the framework.

Write your own class and implement BeanPostProcessor. It comes with two methods to run Before and After initialization. They take two arguments - object and string. They also have a return type of object. This lets you make whatever changes are necessary.

We tell Spring about this class by defining it as a Bean in the XML file with just the class definition, with a value of the fully qualified package name. You can have as many as you want.

BeanFactoryPostProcessor

In much the same way, you'll implement the interface in your custom class. You also inform Spring about it by declaring it as a bean in the XML file.

This gets called before the Singletons are initialized. It's a way to override default functionality in the Bean Factory. A common use is to use placeholders for property values. You could create a file to house the initial property values (objconfig.properties as an example.)

In the XML file, you use variable placeholders in the property value attribute.

```
                                  (name in the config file)
<property name="propName" value="${objName.propNameinConfig}" />
```

You need to pull in the class that will look at the properties file as a Bean.
org.springframework.beans.factory.config.PropertyPlaceholderConfigurer

To tell it where to look, this Bean will need a <property> tag.

```
<property name="locations" value="objconfig.properties" />
```

It needs to be in the same folder as the XML config file, or you can use:

```
<property name="locations" value="classpath:objconfig.properties" />
```

Coding to Interfaces

Instead of invoking a Bean through Spring using the class, you can do so by the interface. You pass through which Bean you want by the name of the bean in the getBean() method. (This may be why Casey said the objects all needed controllers.)

Spring Annotations

@Required
Let's Spring know about dependencies, to catch problems at initialization. It requires a Bean post processor in the XML.

@Autowired
This replaces the need to add a property in the Bean in the XML file. It's also a post processor and also needs a post processor Bean.

When you have multiple Beans, naming your Bean the same as the dependency required means there's no further configuration needed in the XML.

First it will look for type. Next it will look for a matching name. Then it will check for qualifiers. In the XML, add a <qualifier> tag in a Bean definition with a value that acts like an alias. In your program, under the Autowired annotation use:

@Qualifier("aliasName")

It however needs the XML namespace in the file which you can copy from the Spring framework config file.

To avoid adding every post processor in your XML you can use the <context> tag in your <beans> tag.

```
<context:annotation-config />
```

JSR-XXX

Most of these notes are around JSR-250, but it's outdated. javax.annotation

@Resource(name="BeanName")
BeanName is the alias. Otherwise, if your Bean has the same name as your dependency, it will find the correct Bean.

@PostConstruct
Notates a method that runs after a Bean is created.

@PreDestroy
Notates a method that runs right before the Bean is destroyed. It needs the registerShutdownHook() method.

Spring Annotation Stereotypes

The main annotations that will be used on your classes.

@Component
Annotation that let's Spring know what classes need Bean creation. It doesn't need to be defined in the XML. The Bean will be named after the class. In order for Spring to know to scan our application for annotations, we need to add the scan context in the XML file.

```
<context:component-scan base-package="packageYouWantScanned" />
```

Service, Controller, Repository are specializations of Component.

@Service
Let's Spring know the class is a service. This is part of Domain-Driven Design. Services house business logic.

@Repository
Let's Spring know what class is a data object. Interacts with databases or data as a general.

@Controller
Let's Spring know the class is a controller. The controller acts as the messenger between the different pieces, services, repositories and views.

These annotations let Spring know what roles your classes are taking. It also provides a level of documentation for your application.

Spring Boot Routing

@GetMapping(“/endPoint”)
@PostMapping
@PutMapping
@DeleteMapping
@RequestMapping

MessageSource

This is very useful for things like multiple languages. Create a .properties file for messages you wish to set in one space. It’s put together in key / value pairs.

You have to make a Bean in the XML file for this with the class from Spring:
context.support.ResourceBundleMessageSource

It also needs a <property> tag of name=”basenames” to know where to find your .properties files.

```
<list>
  <value>nameOfFile</value>
</list>
```

Context has a getMessage() method. It has several signatures. You can Autowire the dependency on the MessageSource Bean to be able to use the getMessage() method. You can add placeholder arguments in your properties file by using 0 indexed curly brackets. {0}

In the getMessage() arguments, you pass an array of objects.

Event Handling

There’s a variety of interfaces that deal with event handling. If you need to trigger methods based on events, look up Spring’s list of possibilities. There’s too many to list here, just know it’s possible.

AOP

Use an aspect configuration. In order to use the AOP namespace in the XML, you need to define the source in the <beans> tag.

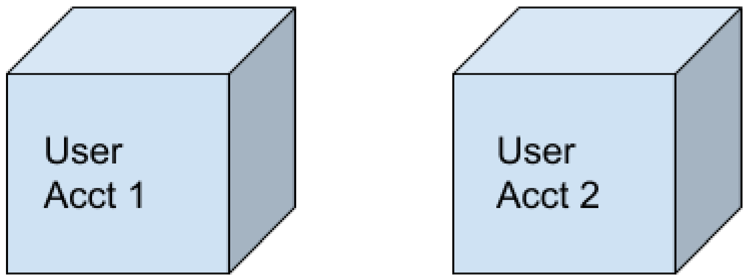
Spring Boot

Takes the Spring framework and abstracts out a lot of the infrastructure setup so you can get to the business logic of your application.

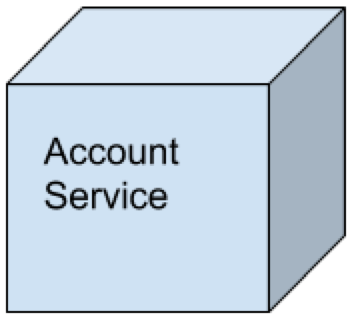
Dependency Injection Design Patterns

Spring is really centered around the idea of Dependency Injection allowing for easily contained code that can be modified without breaking other structures.

It's a layer of abstraction from the instantiation of objects for better code maintenance. Changing object instances should not impact the rest of the code.
By giving over instantiation to Spring (IoC), we also allow Spring to decide how many instances are actually needed. If one instance can serve several parts of the application instead of more, Spring will handle this. Being able to share instances of objects is important in business services.



These objects hold data, therefore, we need more than one.



This however holds no data, so we only need one instance in which the user account can share. For any business service where it only has functionality, you really only want one instance.

Every class declares the dependencies it needs through annotation. Spring will look at these declarations and inject the dependencies to make sure every object has references to every instance they require. Spring also manages the life cycle of the objects.

Inversion of Control

Works towards loose coupling. It's a design principle in OOP.

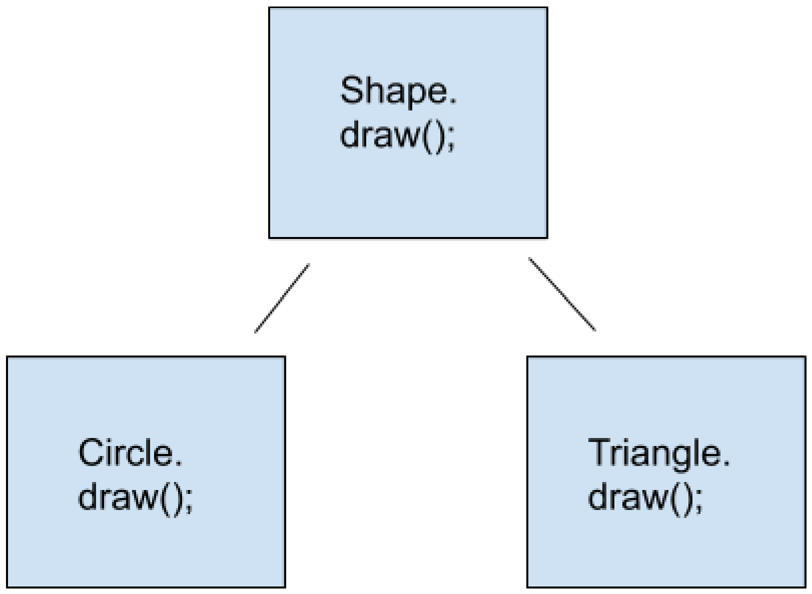
Spring takes over instance creation for a few different reasons. Java classes should be as independent from each other as possible. This increases reusability and makes them easy to test independently. Whenever you use the *new* keyword it creates a hard dependency (hard coupling).

A class shouldn't configure its dependencies statically. This should be configured from the outside and that's what Spring does for us.

Loose Coupling

Hard coupling is when you put the dependency in your class - in short, whenever you use the *new* keyword. By having Dependency Injection containers, Spring manages the creation of instances for you to create this "loose coupling".

Standard Polymorphism:



```
Shape tri = [ new Triangle(); ]
tri.draw();
Shape circ = [ new Circle(); ]
```

```
circ.draw();
```

We're still hard coding an instance of Triangle and Circle, so we don't really benefit from polymorphism here.

A framework class, usually called the Dependency Container, analyzes the dependencies of the class (through the use of annotations). In the analysis, it creates an instance of the class and injects the objects into the defined dependencies (annotations), via Java reflection.

Java Reflection

An API that examines or modifies the behavior of methods at runtime. Using this library we can invoke methods at runtime irrespective of their access modifiers.

A class is built around building the object and it does it with the following methods:

- **Class** - getClass() method finds out what class the object belongs.
- **Constructors** - getConstructors() find the corresponding public constructors.
- **Methods** - getMethods() grabs the public methods of the class.

This is also useful for debugging and testing tools.

Drawbacks:

- **Overhead** - should be avoided in performance sensitive apps
- **Exposure of Internals** - breaks abstractions

Annotation

Annotations are used to describe class dependencies to Spring.

Where dependencies can be injected in a class according to JSR330:

- Constructor
- Fields
- Parameters of a method

Avoid dependency injection on static fields and static methods.

- Static fields are injected after the first class of the object was created via Dependency Injection, so no access in constructor.
- Static fields can't be marked as final or you get errors
- Static methods are called only once after the first instance of the class was created.

Order of Dependency Injection

- constructor
- field

- method

The order in which methods or fields are annotated with @Inject are called is not defined. You can't assume the methods or fields are called in order of their declaration in the class.

AOP

Aspect Oriented Programming

Addresses cross-cutting concerns (code that isn't really business logic, more support code) which is code repeated in different methods and can't normally be refactored in its own module such as logging or verification services. Code that would capture metrics and be repeated through your application can't normally be modularized and so this development methodology tries to do it in a smart way.

Spring uses proxy objects that wrap around the original object and take up the advice that is relevant to the method call. They can be created through a proxy factory Bean manually or through auto proxy configuration in the XML.

It then gets destroyed when the execution completes. Proxy objects are used to enrich the original behavior of the real object.

This helps reduce complexity and boilerplate code.

Aspect: The class that implements the cross-cutting concerns. Configured with XML or annotated with @Aspect. A class with special privileges.

Weaving: Process of linking Aspects with Advised Object. Can be done at load time, compile time or at runtime. Spring does it at runtime.

Advice: Advice are kind of like triggers. The job meant to be done by an Aspect or the action taken by the Aspect at a particular point. There are 5 types:
1. @Before -> before method is invoked
2. @After -> after method is invoked
3. @Around -> bubbles the method
4. @AfterThrowing -> runs after runtime exception
5. @AfterReturning -> after successful method

JoinPoints: An app has thousands of opportunities or points to apply Advice. They're known as join points. Example: Advice can be applied at every invocation of a method or exception be thrown or various other points. Spring AOP supports only method execution join points (advising the execution of methods on Spring Beans).

Pointcut: Selected join points where Advice is applied are known as the Pointcut. Use explicit class and method names or RegEx. Reduces repetitive code by writing once and using at multiple points.

JBS

Java Business Service

Spring can take a class and hold onto it (typically using Singletons) and keeps it in its registers. Other classes and controllers can use them.

`@Service`

In your controller you create a private instance variable calling on the service class. Spring creates the object. Annotate this private instance variable with:

`@Autowired`

It lets Spring know about the dependency to inject it into the class. Put it on anything you need Spring to create an instance for.

In order to use variables in the path you need to make two annotations:

```
@RequestMapping("/topics/{id}")
public Topic getTopic(@PathVariable String id){
  // code
}
```

If the variable is a different name than your parameter, you need to annotate it a little different:

```
public Topic getTopic(@PathVariable ("diffName") String id){
  // code
}
```

Convention is to name it the same. To use other methods, you have to specify it in the @RequestMapping annotation.

`@RequestMapping(method=RequestMethod.POST, value="topics")`

In doing a POST request, you need to send, as part of the header, the content type.

Singletons

Objects in which you only need one instance. It will often get referenced a lot throughout your project, but there will only be one.

Creating a Project

There are several ways to setup a Spring Boot project, the standard method we'll use is using the STS tool.

Set JAVA_HOME in .bash_profile

Manually

Step 1: In STS, choose a Maven project. (Might need to find it in the project wizard.) In the pom.xml file, the <parent> block defines the parent of our project. Our project is the child. Spring Boot has setup a lot of stuff in the Spring Boot Startup Parent. It is already configured for Maven which is why we're making our project a child.

Step 2: Define dependencies (Bill of Materials). In a standard Maven project, you would need to define those dependencies manually. Spring Boot, however, has meta dependencies you can add to your file by simply adding Spring Boot as a dependency.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
```

It then creates the Maven dependencies and you can see the JARs downloaded and put into the project.

Step 3: If you get an error, you may need to update the Maven project by right clicking on JRE > Properties. You may also need to update the Java version in the pom.xml file.

```
<properties>
  <java.version>1.8</java.version>
```

Update the project again.

Step 4: From there in src/main/java create your class that will contain main(). Annotate the class with:

```
@SpringBootApplication
```

STS will ask to import what you need. We then pass our class to Spring with the below inside of the main() method:

```
SpringApplication.run(MyClass.class, args);
```

This gives our application everything it needs to run Spring Boot. You can run the app and see the startup console and going to localhost:8080 shows us the default error page because we haven't set anything up yet.

What does it do?

- Sets up default configuration. This can be tweaked, but generally the default is fine.
- Starts Spring application context.
- Performs class path scan.
- Starts embedded Tomcat server. Don't have to setup a server container.
- Annotating your code for specific behaviors (service/controller), Spring scans your code to treat them appropriately.

Add a Controller

- A Java class.
- Marked with annotations.
- Has info about:
    - What URL access triggers it?
    - What method to run when accessed?

Annotate a controller with:

```
@RestController
```

In order for Spring to know what method is called when the controller class is invoked, we annotate the method.

This creates the json for us. This is the default and only for a GET request.

```
@RequestMapping("/url")
```

If Spring needs to configure the server a specific way, it will do that for you. Embedded Tomcat also creates a standalone application. Very useful for a microservice. It takes care of the architecture.

STS

Pretty much is just magic.

Spring Initializr

Follow the online wizard. You can pick what packages you wish to include and there's less to configure than manually creating a Maven project.

Spring Boot CLI

Allows Groovy scripting. Apache Groovy is a scripting language that works with Java. You can do your scripting directly in the CLI.

Have to install it to have access to the command line tool. Since DaVita doesn't use Spring Boot CLI, this is where I'm wrapping up my knowledge.

Customizing Spring Boot

Property file: application.properties
Go to resources folder, create application.properties there.

Example:

```
server.port=8081
management.port=9001
```

You can find configurable properties at Spring Boot Docs. Search for "Common Applications Properties".

Maven

Package dependency manager, much like NPM. Gradle is built on top of Maven. What's nice is that removing a dependency clears out the package from your project as well - unlike NPM.

pom.xml

This is where you setup your Maven dependencies. Each dependency is its own xml block.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  …
</dependencies>
```

This is then done in real time and the JARs download to your project.

Actuator

Maven dependency that has a lot of cool reporting options.

To package a Maven project, go to your project directory:

```
$mvn package
```

Gradle

Gradle is a build tool. It's fast because it doesn't run tests that haven't changed, processing only inputs that have been modified. It's also good at diagnosing performance issues.

1. Gradle uses plugins to become the right toolkit for your project.
2. Gradle's core model is task centric. Based on dependencies, it will create a task graph to execute.
3. Gradle has a lifecycle:
    a. Initialization - Sets up build.
    b. Configuration - Constructs task graph.
    c. Execution - Runs the tasks.

Avoid expensive work during the configuration phase, because it has to evaluate the code every single time.

DSL

Domain Specific Language. This is a language that works with the JVM and Gradle uses two: Groovy and Kotlin. Their syntax is similar and they can be used interchangeably. It's often easier to work with than XML.

Gradle Wrapper is code for downloading a Gradle distribution. It grabs everything you need to run Gradle inside of your project without needing to install it on your machine and change the environment variable. The configuration can be tweaked, but generally the default is fine.

Gradle supports Maven and Ivy (which is an extension of Ant). Good at supporting multi-build projects and can manage other projects like JavaScript or C++. It's highly customizable as a declarative build language. We tell Gradle what we want to happen, not how we want it to happen, letting it do a lot of the work for us.

```
$ ./gradlew build
```

(Runs all)

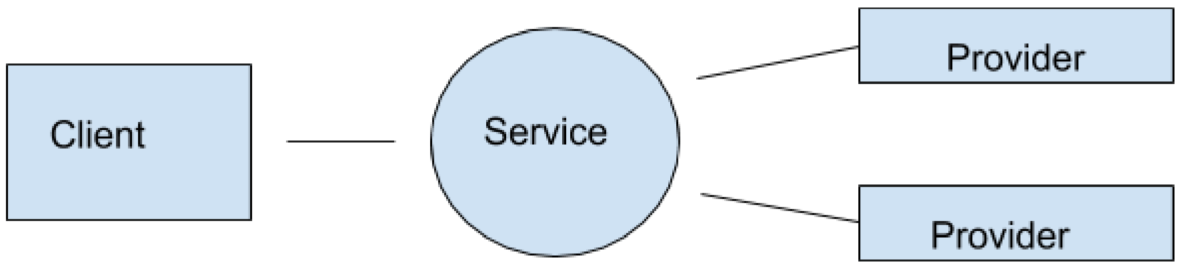
```
$ ./gradlew [task name]
```

(Runs specific task)

Microservices

A Service is specific functionality provided by a library. Apps and libraries providing implementation are Service Providers and the apps using them are called Service Consumers or Clients. It's composed of a set of interfaces and classes and has an interface or abstract class that defines functionality.

Service Loader(s) class is responsible for discovering and loading service providers at runtime for a service interface of type S. Allows for decoupling between the Providers and Consumers. Also allows for Providers to be added or removed without affecting the Service Interface.



In other words, apps over time, became complex. SOP was to modularize, but no one cared about the execution of the code on the client machine.

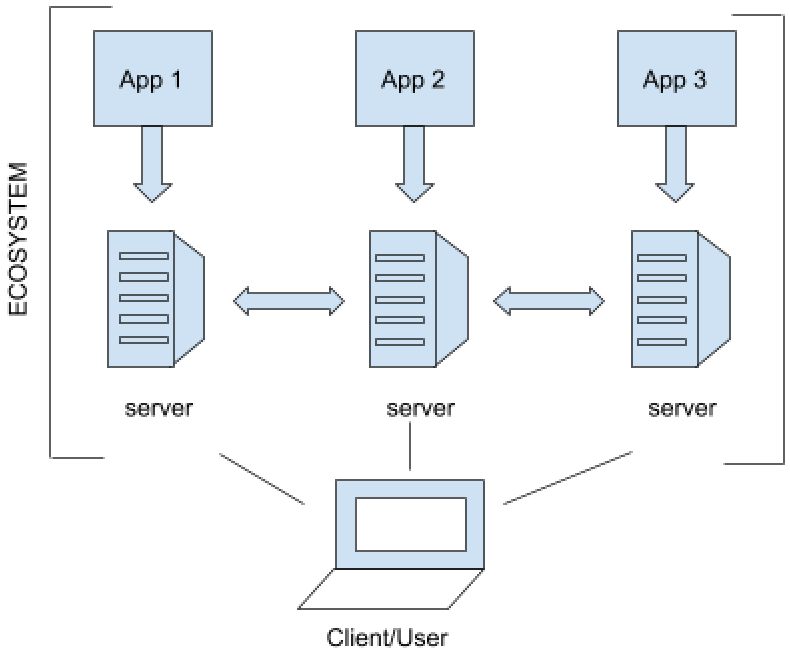
Apps have moved away from standalone installation on the client side and instead are being installed server side to serve to client requests. Still, the execution was ignored.

As the functionality of many applications continue to grow in complexity and need to run almost instantaneously, execution (runtime) has become much more important.

There are several disadvantages to monolithic application architecture:

- Whenever changes are made, the entire application needs to be tested.
- Scalability
- When traffic spikes, new server instances are setup to help deal with the load. The problem is needing to put the whole applications on a server instance, when perhaps users are only using the shopping cart capability, a small section of the code, it creates a lot of overhead.

Breaking up an application:



Even the view can be its own application. They call each other's REST APIs.

Advantages of a Microservice:

- Maintenance is easier, testing is easier, extending is easier.
- Deployment flexibility. Can even use different languages and platforms.
- Scaled separately, reducing overhead.

Disadvantages of a Microservice:

- Possible increased complexity of deployment and architecture.
- Services discovery
- Network issues causing interruption in microservices.

Not all applications need this level of complexity.

REST API

REST - architecture style for designing networked applications.

REpresentational
State
Transfer

Guiding Constraints

1. Client-Server
   a. Separate UI concerns from data storage concerns
2. Stateless

    a. Session state stored entirely on client
3. Cacheable
    a. Right to reuse response data
4. Uniform Interface
    a. Identification of resources
    b. Manipulation of resources through representations
    c. Self descriptive messages
    d. Hypermedia (Hypertext) as the engine of application state
5. Layered System
    a. Components are self contained and cannot “see” beyond the layer they are interacting.
6. Code on Demand (optional)
    a. REST allows client functionality to be extended by applets or scripts

REST != HTTP

REST uses HTTP, generally in the form of microservices, returning most often json objects, but can also return other formats.

Endpoints

GET /topics	Gets all topics
GET /topics/id	Gets a topic
POST /topics	Create new topic
PUT /topics/id	Updates the topic
DELETE /topics/id	Deletes the topic

Actuator

Maven dependency. Gives some additional functionality to the project. Gives new API end points.

http://localhost:8080/health

Gives the status of the environment. You change the management port in the application.properties file. This is useful for setting up different access points.

```
management.port=9001
```

http://localhost:9001/health

There’s a lot of useful information Actuator can give about the app.

Unit Testing

A unit test verifies an assumption about the behavior of a system. You can write tests for classes, methods and functions - the “smallest” pieces of a program.

Unit tests support the following:

- Quick feedback. (Fail fast.)
- Automated regression checking. The tests we write over time can continue to be used and evolve themselves.
- Design aid. Tests act as callers and we can see if the way we call and use the code makes sense. Code that is hard to test can be an early indicator of a design problem.
- Improves confidence with the system.
- Documentation.

Good unit testing needs these qualities:

- Should be automated.
- Fast execution. If it takes too long, developers are less likely to test.
- Tests should be self contained:
    - Don’t depend on other tests.
    - Don’t depend on outside access.
- Be consistent and time and location transparent. Should not fail in other time zones or say, after midnight.
- Tests should be meaningful.
- Tests are system documentation.
- Tests should be short and maintained. They also need refactoring.

Add slow tests to their own automation that can run in the background without having a lot of impact.

SRP

Single Responsibility Principle. A unit of code should do one thing and do that one thing well. Methods need to take input and give output, that’s their job. If output is sent somewhere else before the method is finished executing, like a report or a file, it can be quite difficult to test.

Same can happen with inaccessible inputs. Mixed UI code with business logic is an example of this.

Organization

We organize unit tests by mirroring the source code.

TDD

Test Driven Development. Write your tests first before you start coding. Assert what the feature should do, design the test around the expected execution, and then actually implement the feature. This makes back and forth between production and testing a regular occurence and we can find errors faster.

This helps to produce clear interfaces. A unit test is a client of the code. We can see how clear and easy it is to call our code before any other clients do, such as other APIs. Also helps to produce clean code. It makes refactoring easier as you can be certain of what passes and fails your tests.

Red, Green, Refactor

You have a failing unit test, you implement just enough code to get it to pass. Refactor to clean up the implementing code. Repeat the cycle by adding just enough code to get the test to fail.

Example:

> Setup a new test case and decide on the behavior you wish to test. Once you settle on a name for the method you're testing, the IDE will highlight it in red, because the method doesn't exist, so now you need to write just enough code to get the test to pass. In this case, we can let the IDE create the method stub for us. It will return null.
>
> We'll go back to the test, see it passes, so we keep writing to assert the behavior until it fails again. We'll likely write an assertion to start because we're receiving null, the test will fail. You can add in a basic data type you need in order to get the test to pass. This sort of back and forth is called "Fake it Until You Make it".

Once you've completed the feature and it's passing tests, you can wire it into a UI.

Your code will get complex after a while and your back and forth will decrease in how quickly you're switching, but even still, keep your Red, Green, Refactor cycles as short as possible.

BDD

Behavior Driven Development expands on TDD and has a mock heavy approach. It helps in thinking about how code is supposed to behave when design tests. This also helps keep tests in isolation.

JUnit

These notes are largely for JUnit5.

JUnit can be added as a Maven dependency. We let JUnit know what is a test by using the annotation:

```
@Test
```

Test has setup code, code that executes the system under test and assertion code. If you're testing a method that throws an exception, you have to tell JUnit about it.

```
@Test(expected=RuntimeException.class)
```

JUnit comes with built in assertion methods found at:
https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

They're pretty self explanatory.

- assertAll
- assertArrayEquals
- assertEquals
- assertFalse
- assertIterableEquals
- assertLinesMatches
- assertNotEquals
- assertNotNull
- assertNotSame
- assertNull
- assertSame
- assertThrows
- assertTimeout
- assertTimeoutPreemptively
- assertTrue
- fail

You can run a variety of assertions on a piece of code to understand its behavior and design.

All JUnit assertions allow for customized error reporting. Using a lambda for the message return in an assertion means the return only happens upon failure, speeding up tests.

```
assertEquals(result, time, () -> "error message");
```

We use other annotations to setup and tear down tests:

- @BeforeAll - This will run once before the Test methods.
- @BeforeEach - Will run before each Test method.
- @Test - Annotation for a test method.
- @AfterEach - Will run after each Test method.
- @AfterAll - Run once after all tests.

@BeforeAll and @BeforeEach are at the class level and so are static methods. This can be overwritten, but rarely needs it.

Sample Setup

init All executes

```
int executes
test1 executes
tearDown executes

initAll executes
test2 executes
tearDown executes

tearDownAll executes
```

JUnit has:

`@DisplayName(“desc”)`

This lets us get more explicit without having overly long or complicated method names. They appear in the JUnit console after tests are run.

Nested Tests

Create an inner class

```
@Nested
class groupTests{
  Test1
  Test2
}
```

This can help us provide context to tests that do much the same, but test different aspects of the code.

```
@Nested
@DisplayName(“Test unique exception”)

@Test
@DisplayName(“with wrong formatting”)

@Test
@DisplayName(“with varied string case”)
```

We can then combine our DisplayName annotations to create useful context about what is being tested.

Control Test Method Execution

Sometimes we don't want to run a test, but commenting out blocks of code gets messy and doesn't report itself when we run tests, making them susceptible to being forgotten and cluttering up our code base. We can temporarily disable a test using the Disabled annotation.

```
@Disabled
```

It will show up in the report as ignored and reminds us it still needs to be addressed.

```
assertAll()
```

A way to test multiple things without JUnit stopping at one failed test. It groups similar assertions to get back good reporting all at once.

```
@Tag("my tag")
```

Allows for tests to be run by a tag. You can configure this in the Run configuration submenu. Maven and Gradle can also target tags for testing.

Make Existing Code Testable

Two common problems with tackling existing code:

1. Code with mixed concerns. It's not following SRP.
2. Problematic dependencies

We deal with code with mixed concerns by refactoring - separating concerns. Most IDEs are also built with helpful tools. You can do this slowly over time by extracting the logic you want to test and put it into its own method or class (depending on complexity).

We deal with dependencies by refactoring with Dependency Injection.

MOCKITO

Mockito is a mocking framework for writing unit tests in Java. Mocking allows us to keep unit tests encapsulated and make unit testing overall more effective.

When unit testing we want to focus on the functionality of the unit itself, not its dependencies. The problem is units depend on other units to work. Mocking those dependencies lets us test a single unit without needing to chain test through the entire app.

Test Doubles

- Dummy - Objects passed around but not actually used.

- Fake - Objects have working implementations, but use shortcuts not good for production. (In memory database as an example of this double.)
- Stubs - Canned answers to calls.
- Spies - Stubs that record some info based on how they were called. They also can observe actual objects.
- Mocks - Objects preprogrammed with expectations which for a specification of the calls they are expected to receive.

Dummy

Extend a child class from the one you wish to test. All methods throw a RuntimeException. (Added to src and not test?)

Spy

Gets info on a real object, such as how many times an object has been used.

Mocks

Only mocks insist on behavior verification. The other doubles can, and usually do, use state verification. Mocks behavior like other doubles during the exercise phase, but differ in setup and verification.

Mocking encapsulates tests, making them less maintenance heavy.

Dependencies from other code is called a collaborator. Popular collaborators for mocking:
- Network calls
- Database connections
- Email sending
- Randomness: Time is a good example, because we can’t know from moment to moment what will return.

```
MyService myService = mock(MyService.class);
stub(myService.methodCall(“Parameter”)).return(“value”);
```

You can chain mocking:

```
when(methodCall).thenReturn(value1).thenReturn(value2);
```

Write your assertions in order of expected value. When you don’t tell mocks what to return, they return defaults.

Argument Matcher

This is for when we want to mock any of the return type. We're less concerned about what returns as long as the data type matches.

anyInt -> any integer used will return the thenReturn() chained value. This takes the place of hard coding values to make tests more flexible and dynamic.

Stubs

Right tool for the right job. Generally good for testing proper exception throwing. They however can cause problems in regard to their own separate maintenance and overhead. They also don't do dynamic conditions without getting complex. They're much better for simple problems.

Integration Testing

Combine modules in the application and test as a group to see they're working or not. Carried out after Unit Testing and before System Testing.

Unlike Unit Tests, integration testing check for external dependencies - usually executed by a test team. Maintenance is expensive.

`@WebMvcTest`

Usually used for integration testing, generally on controllers.

In Integration Testing, you want to test one layer at a time, so while testing compatibility between two modules, you'll want to mock other dependencies.

JPA

Java Persistence API is a specification for ORM standards.

Java applications often need to connect to a database. The standard way is with JDBC and it's not easy to work with. Spring uses JPA as Spring JPA to make this connection and management a lot easier.

CRUD

Create Read Update Delete

Making an interface that extends CrudRepository gives access to the JPA CRUD methods.

```
CrudRepository<EntityName, idReturnType>
```

ORM

Object Relational Mapping

ORM lets you map your entity classes into sql tables. It handles the conversion of sql queries to object instances.

Hibernate and Spring JPA both use JPA specification. Hibernate is a JPA "provider". Spring JPA adds an extra layer on top of a provider.

Entity Mapping

Annotate before a Java class and it lets the JPA know it needs to pay attention.

`@Entity`

Adding the Entity means JPA is going to create a table with your instance variables, named after your class.

Add an annotation for the primary key:

`@Id`

CRUD Operations

When you build a data service, you're often doing the same operations over and over again for your entities. Spring has a CRUD repository that can do those basic operations for you. When you build your repository, just extend CrudRepository which needs the Generic defined. The annotation is:

`@ManyToOne`

Let's JPA know the relationship type.

Naming conventions

By using naming conventions for the JPA, it can figure out what sort of queries you're implying. We combine an Introducer, Filter, and Condition if needed.

Introducer

- find
- read
- query
- count
- get

Filter

- Distinct
- First
- Top

Use these to remove duplicates or limit the result set.

Conditions

The criteria contains the entity-specific condition expressions of the query. We can use condition keywords along with the entity's property names. We can also concatenate the expression with And and Or.

- Is
- Equals
- IsNot
- IsNull
- IsNotNull
- ActiveTrue
- ActiveFalse
- StartingWith
- EndingWith
- Containing
- Like
- Between
- In
- LessThan
- LessThanEqual
- GreaterThan
- GreaterThanEqual
- DateAfter
- DateBefore

findByName**Or**BirthDate**And**Active

Precedence is And, then Or. For long complex queries, look at the annotation.

```
@Query
```

Introducers + filters(optional) + conditions

```
@PersistenceContext
```

Spring Data JPA

Enhances JPA and aims to simplify the code while still being feature rich. It also has a repository generator. It uses Query DSL (Domain Specific Language) which allows the creation of the Java Interface Methods, utilizes certain keywords, along with JPA attributes needed to quickly implement queries without too much code. It also works if you need to use other solutions, “getting out of the way”.

To add Spring Data JPA to a project, add the Maven dependency. You also need to add to your application a context file for Spring:

```
<jpa:repositories base-package = “repo.package.name” />
```

You then need to add the xmlns (namespace for JPA).

```
xmlns:jpa=”http://www.springframework.org/schema/data/jpa”
```

In the locations section it needs the same URL above, but also the .xsd file reference.

```
http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
```

Annotation

```
@EnableJpaRepositories
```

Packaging/Distributing

Manual

Put .java and compiled .class files in different folders. (A good IDE will do this when you create the project.) You can tell the compiler where to send compiled files.

If need to ship Java with your program, you can find a Java installer to check if Java is installed and it's the right version.

JAR

Java ARchive

Packaged files, like a zip, with compressed .class, audio, image and directories. Zip tools can also unpack JARs.

JARs can be turned into executables with a manifest file that informs where the main() method is. Use a JAR tool to create the JAR file.

manifest.txt

Place in with classes to inform the computer where the main() method is.

```
Main-class: MyApp
```

The file structure needs to match the file structure you have for your package name in the manifest.txt file if you're doing this manually.

Java Web Start

Launch your app from a browser and it gets downloaded to the user's machine. It works like a browser plugin and its main purpose is to manage the downloading, updating, and launching (executing) of your JWS apps.

Spring

```
$ mvn clean install
```

Clean removes directory if it exists.

Creates a .jar in the project > directory/target/filename.jar

To run your program

```
$ java -jar target/filename.jar
```

Do this in your project directory. -jar can be changed to -war

.jar files > contained java program with standard files.
.war files > web application including jsp, html, JavaScript, etc

API Design

Build API URIs as human readable by using pluralized nouns. When possible, avoid nesting child resources more than once. Lean on locating resources at the root path. You can use urls as a parameter.

Don't do this:

`/orgs/{org_id}/apps/{app_id}/dynos/{dyno_id}`

Do this:

```
/orgs/{org_id}
/orgs/{org_id}/apps
/apps/{app_id}
/apps/{app_id}/dynos
/dynos/{dyno_id}
```

Avoid overly long URLs.

Idempotence

Idempotence is anything with no difference in ending result. In an API, you can have an operation that makes multiple requests that give the same result, thereby acting as a single request. Several http method calls are considered idempotent: GET, HEAD, OPTIONS, TRACE, and PUT. Sometimes DELETE can be considered idempotent, but it's tricky.

It's important to know which ones are idempotent for optimization and performance.

RESTful Methods (HTTP)

GET	HEAD
PUT	PATCH
DELETE	OPTIONS
POST	

You have to specify what options are available in your API (Spring annotation), but then an OPTIONS request will list what's allowed.

PATCH is a bit of a tricky method in that if there's a call to a nonexistent resource, it will be treated as a "create", and a call to an existing resources is considered an update. To change these defaults, you may specify precondition HTTP headers in the request.

PUT is defined as a complete replacement of a resource, so not recommended for modifying resources as you would have to redefine properties you're not modifying.

Queries

You can use filters as queries in your API. There's a type Filter for a class.

Pagination

Spring Boot has a mapping annotation to add pagination as a parameter.

Optimization Principles

STATUS REQUEST can give you information on the resource server.

ERRORS is a way to implement fast failing request validation to protect a service from resource-intensive requests.

Another optimization trick is to return empty values, not 404s. Don't expect more than you think needs to be exposed. Be consistent with results.

DTO - Data Transfer Objects are proxy objects that help decouple your persistence layer from your API. You choose what attributes are accessible.

Data should assumed to be bad until it's been through some form of validation.

Microservices Patterns

API Composition:

API exposure block to perform split joins against one or many downstream service endpoints. Instead of a consumer having to make multiple queries, a single request payload can be split into multiple downstream calls. The responses can then be joined into a single payload before being sent to the consumer.

Redaction

Ability to remove, mask, or limit the presence of fields in request/response payloads or headers. (IE, credit card info.)

Push Notifications

Normally the client has to poll the data for changes, which is inefficient. Using webhooks enables API consumers to subscribe to specific API events. API consumers provide a callback URL that is used by the server to push the event.

Service Mesh

A smart inter-service communication infrastructure which decouples and prevents faults from being propagated.

A Service Mesh should:

- Reliable, efficient, secured, and fast transport infrastructure that ensures services always reach one another.
- Dynamically discover new services through a registry.
- Dynamically route and load-balance requests to all or specific active instances of a service. (Can do a portion of traffic.)
- Prevent faults from being propagated by isolating underperforming or malfunctioning service instances.

A service mesh is typically delivered as part of an independent registry.

Event Hub

Asynchronous events(choregraphies) that can be subscribed to.

Event Hubs should:

- Event store - persist events
- Reliable messaging
- Publish and subscribe to events
- Connectors - natively connect multiple sources or targets
- Event processing - introspect in real time to detect patterns in the inflow of data.

Apache Kafka is a popular technology to implement an Event Hub.

Service Registry

Key/Value pair storage used in fully decoupled service infrastructures to store runtime and configuration metadata. Can be used to dynamically determine the status of a service and route requests to active and healthy endpoints.

Service Registries should:

- Resource registration - register a service/API endpoint by calling a registry's management REST API.

- Resource Discovery - discover registered resources, including status and health, for load balancing.
- Key/Value Storage - key value store accessed by API proxies or load balancers.
- Key/Value Replication - ability to replicate key value store across different instances of a service registry
- Resource Health Check - services continuously provide status checks and consumers can query that status of services and their endpoints.
- Secret Vault - a secret is anything that requires tightly controlled access. A vault provides a unified interface with tight access, recording, and detailed audit log.

API Gateway

Takes calls from a client, invokes multiple microservices and aggregates the results. It's responsible for request routing, composition, and protocol translation. If there are failed services, the gateway can mask them by returning cached or default data.

HATEOAS

Hypermedia As The Engine Of Application State. Uses a links block to have relative paths enabled.

```
{
    "departmentId": 10,
    "departmentName": "Administration",
    "locationId": 1700,
    "managerId": 200,
    "links": [
        {
            "href": "10/employees",
            "rel": "employees",
            "type" : "GET"
        }
    ]
}
```

Documentation

Things that developers have said help with Developer Experience (Dx):

- Add HTTP verbs support, include examples of requests and response payloads.
- Error handling and what errors to expect.
- Description of any constraints.
- Information on how the API handles authentication and authorization.
- Comprehensive overview.

- Onboarding - how to start using the API. May involve a registration process, capture user details, select monetization scheme, payment options, user credentials, and application key.

Swagger

Swagger is an API scaffolding tool to help with design and documentation. It has its own short form language to quickly template end-point expected results. To add Swagger to a project, you can use Spring Boot annotations. It will need to be added as a Maven dependency via Springfox and Springfox UI. It will also require a variety of other libraries to support Spring Boot documentation.

Reusable Components:
- Definitions
- Responses
- Parameters

Definition:
The entity schema of your object. You can add a definition block for your object at the end of your API schema and just reference it in your http response codes.

```
$ref: '#/definitions/Object'

…

definitions:
  Object:
    type:object
    properties:
      prop1:
        type:string
      prop2:
        type:integer
```

You can also do this across end-points with similar parameters, such as PageLimit or PageNumber. This is also handy for end-points that share response codes.

Configuration

Make a Spring Bean of type Docket.

```
@Bean
public Docket swaggerConfig{
  return new Docket(DocumentationType.SWAGGER_2)
    .select()
    .paths(PathSelectors ant ("/api/*"))
  .apis(RequestHandlerSelectors.basePackage("myPack"))
  .build();
}
```

To add metadata:

```
@Bean
public Docket swaggerConfig{
  return new Docket(DocumentationType.SWAGGER_2)
    .select()
    .paths(PathSelectors ant ("/api/*"))
  .apis(RequestHandlerSelectors.basePackage("myPack"))
  .build()
  .apiInfo(apiDetails());
}

private ApiInfo apiDetails(){
  return new ApiInfo(
    "api name",
    "desc",
    "version",
    …)
}
```

Adding details - use Spring annotation:

```
@ApiOperation(
  value = "Title",
  notes = "Info",
  response = Model.class)
```

There are a lot of annotations you can use for Swagger, look them up in the documentation.

Docker

Virtual machines spin up an entire OS which can be heavy on the host machine. Docker uses the host machine kernel instead, keeping the containers lightweight. Because Linux based OSs use the same kernel, this is possible. This means it uses less disk space and memory and is faster than a virtual machine.

Docker containers use images that bundle the OS, software, and application. It keeps this information in a Docker file (plain text) with a list of steps to perform to create that image.

The community has a repository of images. Official ones are the best. Docker layers images together, separating concerns.

Volumes

1. Persist and share containers and share data between containers.
2. Share folders between the host and the container

It's initialized with a -v flag when you use the Docker run command. The -p flag is the port. Use the full path.

```
docker run -p 80:80 -v /full/path/local/machine/:/var/www/html/
```

Once you're done developing, you'll have to rebuild the image.

It's important to separate concerns in Docker so you don't have other processes in the container when the main process finishes and the container closes, prematurely shutting down processes.

To login:

```
sudo docker login
```

Pull an image from Docker Hub:

```
sudo docker pull [image]
```

You can create an .env_list file and place any environment variables there.

To run:

```
docker run nameofcontainer
```