IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Volume-4, Special Issue-5, Jul 2016 E-ISSN: 2347-2693

Twelve-Factor application pattern with Spring Framework

Dinkar Thakur[1*], Arvind Kalia[2]

[1]*Department of Computer Science, Himachal Pradesh University, India*
[2]*Professor, Department of Computer Science, Himachal Pradesh University, India*

Available online at: www.ijcseonline.org

***Abstract*—** The world of software development is moving from monoliths to micro-service architecture. Instead of having one big application, handling all tasks, there is a paradigm shift towards creating many small applications doing just a unit of work. When we build software-as-a-service we have to look into development, maintenance, and scalability of the application. Over the time it has been observed that the post development tasks of an application are inhibited because application is not just doing one task that it is supposed to do but all the activities that can be delegated. With twelve-factor pattern we can come close to create micro-services that can reduce cost of development, have maximum portability when deploying on different environment, are cloud ready and most importantly scale up. Spring.io provide a rich underlying framework, which can help in creating applications that are twelve-factor compliance, with Spring Boot they took it to a whole new level. With twelve-factor methodology in mind, in this research paper we will try to look into how spring can help covering each point in twelve-factors.

***Index Terms*—**Twelve-factor application; Cloud native; Spring-framework; Micro-services

I. INTRODUCTION

The twelve-factor[1]application pattern is a new way of creating software application such that the applications created by following this methodology ease the process of deploying, maintaining, and scaling and graceful degradation. In short application created with these factors in mind are close to the cloud native application.

A cloud native application can be packaged in containers, allow dynamic scaling and most importantly micro-service based [1]. With the advancement of the cloud computing, there is more emphasis on creating software that can take full advantage of the hardware and services provided by the cloud [2]. In an ideal cloud environment every application behaves as a service, which can be attached from outside of the consumer application and hence are cloud native. The benefit of creating such application is that, these services can utilize other services being provided by the cloud platforms such as Amazon, Azure, and Cloud Foundry to name a few.

The twelve factor methodology provides some guidelines, checkpoint to ensure that when we create an application for the cloud we know what things we have to look up, and what points we need to keep in mind so that we are not developing application every time we need to deploy or changing configuration. Like all the patterns and methodologies its not compulsory to follow these guidelines to run an application in the cloud, but to make full use of the cloud, knowingly or unknowingly we will be following these.

With the evolution of software applications from simple hello-world to complex booking or stock trading application, frameworks facilitating software development have also been improved. One such framework for Java application is spring framework[2]. Developed by Rod Johnson, first released on 1 October 2002, this framework was at core build on principles of dependency injection and aspect-oriented programing with a unit of work in mind. Fast forward to the age of cloud, developers as spring have evolved this framework to provide a comprehensive tool for developing cloud native applications. From creating to monolith war files, spring with its spring-boot frameworks now created micro-services as fat-jars. Spring-Boot[3] has changed the way we develop Java web application, instead of deploying the war in a tomcat container, now we have a fat-jar containing all the necessary components within even the web servers like tomcat or jetty. Hence making it a good choice to create cloud native applications.

II. TWELVE FACTORS AND SPRING

A. CodeBase: One Code Base, Many Deployments.

Tracking of the changes in a twelve-factor application is done using a version control system. All the changes made in the code create a revision and are maintained by the code repository.Git, Subversion and Mercurial are some examples of version control systems [3].

According to the twelve-factors, one application must correspond to one code base and vice-versa. Meaning that one micro-service should have its one code repository. The need for having this is that the changes in micro-services

[1]http://12factor.net/admin-processes

[2]https://spring.io/
[3]http://projects.spring.io/spring-boot/

must be local to it.

One may question that what is the problem in having one codebase and many applications? Well, that's true in the case of monoliths applications where one application is not doing a single task, but a group of tasks. If we look further, we will find thatthese applications have many independent components, but still are in one application as their code base is same forming an Anti-pattern. The main drawback arising from such structure of having a single code base is that for a change in one component we have to deploy complete application, hence we have to test complete application even the component that are not part of the change [6].

Secondly, if we say that we have many code-base constituting a single application, then each codebase itself constitutes to individual application and they should be in compliance with the twelve factors guidelines. Such an arrangement of the code will result in distributed structure and consumption of these applications/micro-services can be either by APIs or by using dependency management systems like maven or ruby gems.

The application can have many deployments, meaning hereby that application might be deployed many times in cloud as separate instances of the same service. Even while under development there might be different instances of the application running on staging environment, on developers machine all referring to same codebase but different version maintained by the version control system.

With spring boot providing the needed underlying framework through dependency management, and passing configuration as environment variable, building and deploying application is just one command. All we need to do is checkout the code, built it and run. No changes other than environment variables like database path, log path are required, these changes are provided by the springconfiguration server. So, we have one codebase per application/service and many deployments.

B. Dependencies: Explicit and Isolated Dependencies.

No application can run in isolation and it depends on the libraries. Some libraries are provided by the language runtime like the system library in Java and the header files in C, but there are many other libraries that don't come as a package and are installed explicitly like some database connector, or image processing libraries. In an application following twelve-factor itis required to explicitly declare all the external dependencies. One must not depend on the libraries or packages being provided by the underlying system. Even if the underlying system on which the application has to run provides those libraries.

The main reason for this is that while creating the application for the cloud one cannot always have the same operating system, same environment for execution. A cloud native application must declare its dependencies completely and exactly. Also, it should not allow any implicit dependency to get in during the execution of the application.

One of the key benefits of declaring dependencies explicitly is that all we need to do is check out the code and run package/builder command and dependency management module will download all the libraries required to run [4]. Instead of looking for the code for dependencies developer only have to go though the manifest, like pom.xml, build.gradle in Java or a gemfile in ruby-on-rails.

Spring-boot comes very handy in this situation with all the framework level dependencies being handled by the spring-boot BOM, all we have to do is add the dependency in the dependency management module. The dependency management system will take care of the rest. Using the help of spring BOM (Bill of Material) it will try to import best-suited library version from the repository [5]. Spring also provides starter modules for many components. Once the application has built successfully we can then package that into a single fat-jar containing all the dependencies it needs to work. So spring boot help is overcoming what is commonly known as dependency hell. Care should be taken here to avoid the explicit defining theversion of dependenciesasit might happen that these libraries conflict with one another. So best is to rely on the spring-boot BOM and let the spring decide which libraries to load. We only need to specify the version in case of libraries not covered under BOM, for example our own custom libraries.

It should also be done in case of dependencies that are included in many micro-services of the same application. One may consider creating a parent module and inject the dependencies through that to all the child application. That will in short run solve some problem of maintaining same version, but as the dependencies increase they will create dependency hell as one application is using one version and second application another. If a dependency is the part of the framework, then it can be defined in the parent dependency management file,butthe actual inclusion of the dependency need to be done in the application.

C. Config: Storing Config in the Environment.

As we have seen that the application with twelve-factor methodology will have one codebase and many deployments. And application configuration will vary from machine to machine, stage to stage. For example same application will be running on different port different IP on every developers machine.Most commonly the database connection will change in development, testing and production stages. If we keep on changing the code each time we have to deploy the application then that will work only good as far as manual deployment is concerned.

Deployment in cloud native application is a continuous and automated process all the developer needs to do is just push the code to the code repository. Creating a new release version after running test cases has to be automated process. In such a case if we need to change the code every time the application is deployed, will limit the process. So configuration and the code need to be strictly separated [7]. As there is much difference in the configuration rather that the code in different deployments.

Further in case of cloud deployment with the use of Dockers [8],If the configuration is inside the application and cannot be overridden from the environment then we have to again build the image for the docker, where as all we need is change the configuration variable.In earlier Java application the properties files are used to do this and these property files are placed outside the war so as to change them without re-deploying the complete application, but we still need to restart the application for these configuration to take place. One major drawback of having configuration in the code is that we can't keep these code files containing the configuration in codebase as they might contain sensitive data like database credentials.

Spring Cloud[4] provides spring-cloud-config component, which take care of the entire configuration related issues. Further more the configuration in spring boot application can be provided as runtime parameter or even read from the environment. Spring cloud config provides externalized configuration support for distributed system. The spring cloud config provides support for both client and server side. This component creates a configuration server and all the client will treat is as a service when the application boots up it will ask for its own configuration. Only bootstrap configuration is need in the application that to can be provided by the environment. Once the application gets its configuration from the server the bootstrap configuration are over-ridden by the configuration provided by the configuration server. Also by providing the RefreshScope in the client application, we just need to change the configuration in the server and configuration server will push the changes to the client application while its still running so using spring cloud config component, externalization of the configuration can be done in real sense.

D. Backing Services: Services as Attached Resourses.

Any service which the application consumes over the network for its normal operation is called backing service. For example database, queuing system, mailing service. These all act as back end services for the application. The application should not depend upon these services for running. There can be some exceptions being thrown if application fails to find a backing service it requires, but the application must degrade gracefully. There may be services those are not native to the application like the metrics gathering services or third party social networking APIs like Facebook or Twitter. An application following the twelve factors treats these local or third party services as backing services. At any give time the application must be able to swap between these services, without any code change. For example, instead Oracle database the application can be made to run on MySQL cluster, instead of RabbitMQ[5], one should be able to switch to ActiveMQ[6].

Wewill be able to switch backing services if they are treated as resources being provided by the cloud platform. One benefit of having services are resource is that they can be scaled up or down as required, and the application need not to bother about that. We can change the underlying service without and change in the code of the application and the application will behave as it is.

Spring boot provides wrappers for nearly all the known required services, for database we have spring data, for message queues we have the spring cloud stream, for social networking API we have spring social and all these can be bundled in micro-services using spring boot. We only need to change the configuration, through the configuration server in order to make it work. As spring provides wrapper for these backing services as a developer one only need to follow the guidelines provided by the spring framework. Also, if there is a need to create any custom implementation forbacking services,then it must be kept in mind to treat services as resources present outside of the application and accessed through theURL.

E. Build, Release, Run: Seperation between Build and Run Stages.

Build stage transforms the code from the code base to executablelibraries. Dependencies are fetched depending upon the version described in the code base. Build stage does all the hard work. Release Stage creates combination of build produced and the configuration as needed by the staging environment and makes it ready for the execution. In the Run stage the application is made available on the server. The run stage should be the lightest and cleanest part of the process. All the issues should be handled in the build stage as the developers handle this stage. If the server crashes or the application start misbehaving then the application should be able to restart without human intervention. Thus the run stage should have minimum moving parts.

The concept of fat-jars of spring boot comes very handy in these cases. All the dependencies declared are explicitly

[4]http://projects.spring.io/spring-cloud/

[5]https://www.rabbitmq.com/

[6]http://activemq.apache.org/

declared in the application and during the build phase the all these dependencies are fetched and bundled with theapplication. This build of the application is ready to be deployed and will get configuration from the configuration server. With the advancement of Dockers now the image of these build can be saved and run on demand. Many images of different versions of the application can be made which facilitated the rollback and re-deployment of the application.

F. Processes: Application as Stateless and Share-Nothing.

This is the most important concept when we are developing an application for the cloud. When we say build for failure this is what we have in mind. In cloud environment, we have many instances of the same application running on either same machine inside dockers or in different machine serving many requests. If the application is not stateless,it means that the current request depend upon the previous request, then this will lead to failure as the requests can go to any instance and not to the same instance every time.

Any data that need to be persisted between the calls can be shared by using backing services that can be a database or cache server, which is shared across the applications and all the instances. Although, we can use local memory, file system for temporary storage for a single transaction. The result of these operations must be stored in the database or intermediate service to provide persistence. As we will move forward, with each request the data will also move from the state of inconsistency to eventual consistency [9].

Some web systems use sticky sessions, storing the request information in the server. This was old approach where data is stored in the session variables, which in turn get stored on the server. But such applications breaks when placed behind the load balancer, as concurrent requests usually don't go to the same instance.

Micro-services created with spring boot actas different processes and can be consumed by http rest calling, which itself is a stateless protocol. Spring provides spring session component to which act as an external resource shared between the application instances to maintain and provide the session to all the instances. Spring session is mainly used to share the user login session across the micro-services. Extra information and intermediate data can be stored in Redis server or Memcache for which spring has pre-build component all we need to do is include that component in the application and provide configuration in the configuration server for that application. Even though spring provides the component to handle this, it's the developer who needs to keep in mind this concept of statelessness and share-nothing while developing an application for the cloud.

G. Port Binding: Export Services via Port.

Port binding enables the application to become a backing service to any other application. This relates to the concept that the applications should be built as they are backing services being consumed by the URL. With the use of port binding applications can consume by the other application on the cloud as a service. Making applications self-containing by explicitly declaring the dependencies it require inside the application to fulfill this. Traditionally we have to deploy web application in web server, like Java war is deployed in tomcat, PHP uses apache server and ASP.net uses IIS.

Twelve-factor app does not depend on the runtime, in-fact the web-server is also included in the application and with port binding it is ready to listen for the incoming requests. With having an embedded server in the application any type of service can be exposed using port binding.

We can use spring discovery component (Eureka or Consul). All the application can be made discoverable to the other services using this. When an application boots-up its register itself to the spring discovery and when service is requested, the discovery service provides the necessary URL for the service. This concept of port binding is very useful when application is running inside a docker asdocker will bind an internal port to an external port.

H. Concurrency: Scale Out via Process Model.

Processes are the first class citizen in Twelve-Factor. This factor has been derived from the UNIX process model of running service daemons. This means that all the application/processes should scale, clone and restart when required. These processes are handled by the operating system's process manager, which manages output streams and manual shutdown and restarts. Using this model we can design our application so that each process is specialized in handling one type of work effectively. Like the web request are handled by the web worker process and heavy background task by the backing worker process.

If we look application as a whole we will see that an application is nothing but bundle of these micro-services, for example, in case of a website, signup/registration can be one service. Logging is another services and maintaining session another. If these services are combined in a single application like in a monolith application, then scaling an individual component will be next to impossible. We have to scale whole of the application if we need to scale anyone of the lagging component. So if we need to scale out, then we need to have a process model, which can only be possible if the applications are treated as backing services independent of one another [10]. This is highly facilitated by the fact that we create self-contained fata-jars, so that they are totally independent.

I. Disposability: Maximun Robustness, Fast Startup, Graceful Shutdown.

The application should be able to start immediately and when it goes down there should not be like sudden death. If we take an example of a web application, and we follow all the abovefactors even then we can't we certain that the application be behave nicely on web scale. If a backing service goes down, then what should be the behavior of the application? Should it also crash, give an exception or perform some notification.

First thing here is the robustness, it doesn't mean that the application should never crash. There are many factors, which can cause the application to crash. One of them is the failure of underlying hardware. In this case there is not way the service itself can broadcast the panic message. Here the use of the queuing system can help in achieving the robustness. If the backing service fails, then the client should be able to create the queue for the upcoming request. Again this should not be done inside the client or service, but outside of both the application, in a separate queuing server.

Second, the fast startup means that the application can be instantiated within seconds. There should not be any delay in the starting up of new instance of the application. The reason is quite simple when the load increases on a service, then it is distributed to different instances of the service. In order to do so the cloud usually tries to replicate the service and create new instances on the service. If this takes time, then the already running instances have to bear the extra load till the new instance is created. This is effectively handled by providing pre build application (images in case of dockers) as most of the time is consumed in building the application rather than releasing and running it.

Third and important one is graceful degradation. This means that when the process manager terminates the service,then it should close the port in which they are running, thereby refusing any request coming to that port, and finishing any current request to complete. In this model all the services are reentrant. It should not leave any current task in between. Either the task should be completed or reverted back to the queue to be picked next time removing all the intermediate changes.

Also the client should be made aware that service instance is down and requests are reverted to any live instance of the service. Ifthere is no instance in live state at that given point of time, then either the request is kept in the queue or it should go to the fallback. In case of web application spring hystrix along with the client side load balancing handles it. The discovery service provides only those instances of the application that are in upstate. If the client is not able to send requests to the service, then hystrix handles the fallback and execute the fallback method. A typical example is aboutshowing movies based on the viewing history of the user. If this service is down, then the application calling this should handle it gracefully and respond by providing the default movie list. It is not always wise to show the default data, as the user might be expecting some valid output based on his input, and providing him with default data will be misleading. In these scenarios the application must provide a valid error message instead of just crashing.

J. Dev/Prod Parity: Keeping Development, Staging and Production Environment as Similar as Possible.

Usually there are many gaps in the development environment and the production environment. These are mainly because of the stack present in the development environment can be different than that of production. In development phase a developer may use alightweight and fast service like SQLite, and on production dev-ops team would like to go for more robust and heavyweight stack like MySQL. Even the underlying operating system can be different. Developer might go for the OSX for its ease of use, where as the production environment uses Linux [12]. Even the people developing the application and who is deploying it are different. This can lead to uncertain problems arising only in production environment. Hence increasing the development to production time.

Continuous deployment is one of the aspects of the Twelve Factor application. This can be achieved by making the gap between the development and production stack as small as possible. By getting the developers involved in the deployment phase monitoring the behavior of the application after deployment, andinvolving the dev-ops team in the decisions like which component to use as they are the one who have to maintain it.

This parity between the different stages is even more in case of backing service like the database, message queuing system. It is recommended to resist the urge to usedifferent backing services between development and production environment. Now days installing many of the backing services like Elastic Search, Memcache, logging framework on development environment is easy. The only need is to have framework provide support to switch between these services without any change in the code, although changes in the configuration are accepted as long as they are provided from the outside of the application.

Spring with its adapters for backing service provide a great relief, with the help of spring data we have the option of moving from one to another database very easily. With spring cloud stream using any famous message queuing system very easy, all we need is to point the application to the new URL. Still care need to be taken to minimize the gap as any tiny issue while deploying and running application will result in rework of many days.

K. Logs: Logs as Event Streams.

The behavior of the running application can be seen using logs. Usually logs are written in a file. Logs are timed order event streams. Different level of logging is used to filter the depth of the events that are happing in the service while it is running. Logs have no fixed end, as they are output stream, which generates as long as the application is running.

Routing or storing of the output stream is not part of the application. Storing and managing the log file is the concern of the underlying platform all the application need to do is to stream them. Open source log router like fluentd[7], logstash[8] and, greylog[9] are handy in such cases, where the application route the log to output stream, which is then captured by the fluent and send for elastic search for long term archival. The purpose of this is the separation of the concern and delegation of the task, which are not native to the application. By parsing the log to the log capturing services we can parse the log study application run time behavior by plotting trends from the logs.

In the cloud applications communicate with one another and hence it is desirable to have request tracing between the calls to see how the calling between different services is happening. Spring provides spring cloud sleuth and spring cloud zipkin to handle these types of problems. With the help of these components we can have logs to trace the request propagation. Also it is required to have live, instantaneous view of the application with spring hystrix and the turbine stream, application can stream the activity log and system administrator can have real time view of the application.

So much importance is given to the logging because it is a heavy process and on production environment. It is the only way to understand the behavior of the application. With the help of better logging we will able to monitor the service better and take necessary step [3]. The logs it generates govern even the auto scaling and the fault tolerance behavior of an application.

L. Admin Processes: Running Management Task Process.

Once the application is in production then the actual work starts. We may have tested the application over and over again with almost identical dataset and nearly same environment as that of production, but still to make improvements in the application we need to gather data on health performance and usage. Along with that we may require cleaning up of the bad data. Doing A/B testing,running database migrations are also some administrative task one would like to do on the production with the service in running state. Admin processes therefore should run as a long running process of the application. They use the same configuration and code base meaning they are run against the same release stage as of the application itself.

To achieve this the admin process task should be shipped with the application, otherwise it may lead to the synchronization issues. It is recommended to have a REPL shell out of the box for application, or itprovides native some admin endpoints. In the case of spring, spring-boot provides actuator, which exposes management endpoints and URLs for different task that can be done. For example is case of discovery service one can mark the application down so that the application frees the ports and no longer accept the incoming requests. With all the metrics being provided by spring, we can see service health status, JVM metrics, memory metrics, view environment variables [13]. These tasks may appear worthless in discussion, but these tasks are actually very important when dealing with live application.

III. CONCLUSION

When creating an application for the cloud we have to come out of the traditional approach of creating software. With the change in how the cloud platform handles deployment, execution, maintenance and scalability of the application, we have to change the way application are developed to make them more cloud native. In this research paper we have triedto understand the factors that can help developers to create application that are cloud ready.

We also discussed the problems that may arise in an application that is not in compliance with the twelve-factors.We also look into the components provided by the spring framework, and try to map the components to the factors. We find that using the modules provided by the spring framework we could easily make our application a twelve-factor application. Instead of worrying about writing custom components to address the needs of each factor with the use of spring we can invest our time in creating the application rather that the complete framework.

It should be noted that these factors only provide a methodology, a guideline to create an application. It is not compulsory to follow these points. We can still create and run an application on the cloud even if it does not go with the twelve-factors. But as we know that developing of an application is just a tip of iceberg actual work starts after the application is live and running at web scale. If we consider these factors while developing the application, then life of all of us will be easy once the application is deployed.

ACKNOWLEDGMENT

I would like to thank the open-source community to provide with the tools that I have discussed here. I also thank Dr.

[7]http://www.fluentd.org/
[8]https://www.elastic.co/products/logstash
[9]https://www.graylog.org/

Arvind Kalia, Professor, Department of Computer Science, HPU Shimla. I would also like to thank my friends Mrs. Neetika Sharma and Mr. Varinder Singh to help me in the completion of this research paper.

REFERENCES

[1] S. Newman, "Building Microservices". O'Reilly, **2015**
[2] V.Andrikopoulos, S.Strauch,C. Fehling and F.Leymann, "CAP-Oriented Design for Cloud-native Applications"
[3] S.Otte, "Version Control Systems." Computer Systems and Telematics, Institute of Computer Science, FreieUniversität, Berlin, Germany**2009**.
[4] N.Sangal, E. Jordan, V.Sinha, and D..Jackson "Using dependency models to manage complex software architecture" in Proceedings of 20th*Annual ACM Conf. Object-Oriented Programming, Systems, Languages,* New York, **2005**
[5] J.Salecker, and D.Schütz, "Bill of Material" in Proceedings of 9th*European Conference on Pattern Languages of Programs (EuroPLoP 2004)*. UVK, Konstanz, Germany **2005**
[6] Y. Tao, Y. Dang, T.Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? An exploratory study in industry" in Proceeding of *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, North Carolina, November 11-16, **2012**
[7] Maurer, Michael, I.Brandic, and R.Sakellariou, "Adaptive resource configuration for Cloud infrastructure management." *Future Generation Computer Systems* 29, no. 2, **2013,**pp.**472-487**.
[8] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes", *IEEE Cloud Computing*, vol.1, no. 3, Sept. **2014,** pp.**81-84**
[9] M.Hogan, "Shared-Disk vs. Shared-Nothing" - http://www.sersc.org/journals/IJEIC/vol2_Is4/15.pdf
[10] Tanenbaum, S. Andrew, V. Steen, and Maarten, "*Distributed Systems: Principles and Paradigms*" Prentice Hall ISBN 0-13-088893-1
[11] M. Stine,"Migrating to Cloud-Native Application Architectures,"O'Reilly, **2015**
[12] A Modernized Software Environment for Developing, Deploying and Operating Cloud Applications, vmware.http://www.vmware.com/files/pdf/Modernizing_App_ Develpment_Whitepaper.pdf
[13] J.Cito, P.Leitner,, H. C. Gall, A. Dadashi, A. Keller, and A Roth, "Runtime Metric meets Developer - Building better Cloud Applications using Feedback," in Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015), **2015**.

