# What are 12 factor Principal in Microservices Systems

## 2. What Is the Twelve-Factor Methodology?

**The twelve-factor methodology is a set of twelve best practices to develop applications developed to run as a service**. This was originally drafted by Heroku for applications deployed as services on their cloud platform, back in 2011. Over time, this has proved to be generic enough for any [software-as-a-service](https://en.wikipedia.org/wiki/Software_as_a_service) (SaaS) development.

So, what do we mean by software-as-a-service? Traditionally we design, develop, deploy, and maintain software solutions to derive business value from it. But, we don't have to engage in this process to achieve the same result necessarily. For instance, calculating applicable tax is a generic function in many domains.

Now, we may decide to build and manage this service our selves or **subscribe to a commercial service offering**. Such **service offerings are what we know as software-as-a-service**.

While software-as-a-service doesn't impose any restriction on the architecture it's developed on; it's quite useful to adopt some best practices.

If we design our software to be modular, portable, and scalable on modern cloud platforms, it's quite amenable to our service offerings. This is where the twelve-factor methodology helps in. We'll see them in action later in the tutorial.

## 3. Microservice with Spring Boot

[Microservice](https://www.baeldung.com/spring-microservices-guide) is an architectural style to develop software as loosely coupled services. The key requirement here is that the **services should be organized around business domain boundaries**. This is often the most difficult part to identify.

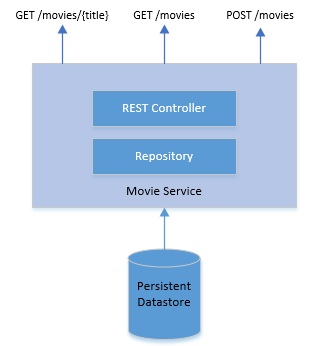
Moreover, a service here has the sole authority over its data and exposes operations to other services. Communication between services is typically over lightweight protocols like HTTP. This results in independently deployable and scalable services.

Now, microservice architecture and software-as-a-service are not dependent on each other. But, it's not difficult to understand that, when **developing software-as-a-service, leveraging the microservice architecture is quite beneficial**. It helps to achieve a lot of goals we discussed earlier, like modularity and scalability.

[Spring Boot](https://spring.io/projects/spring-boot) is an application framework based on Spring which takes away a lot of boilerplate associated with developing an enterprise application. It gives us a highly-opinionated but flexible platform to develop microservices. For this tutorial, we'll leverage Spring Boot to deliver a microservice using the twelve-factor methodology.

## 4. Applying Twelve-Factor Methodology

Let's now define a simple application that we'll try to develop with the tools and practices we just discussed. We all love watching movies, but it's challenging to keep track of the movies we've already watched.

Now, who would like to start a movie and then abandon it later? What we need is a simple service to record and query movies that we've watched:[](https://www.baeldung.com/wp-content/uploads/2019/09/12-factpr-app.jpg)

This is quite a simple and standard microservice with a data store and REST endpoints. We need to define a model which will map to persistence as well:

@Entity

**public** **class** **Movie** {

@Id

**private** Long id;

**private** String title;

**private** String year;

**private** String rating;

// getters and setters

}

We've defined a JPA entity with an id and a few other attributes. Let's now see what the REST controller looks like:

@RestController

**public** **class** **MovieController** {

@Autowired

**private** MovieRepository movieRepository;

@GetMapping("/movies")

**public** List<Movie> **retrieveAllStudents**() {

**return** movieRepository.findAll();

}

@GetMapping("/movies/{id}")

**public** Movie **retrieveStudent**(@PathVariable Long id) {

**return** movieRepository.findById(id).get();

}

@PostMapping("/movies")

**public** Long **createStudent**(@RequestBody Movie movie) {

**return** movieRepository.save(movie).getId();

}

}

This covers the base of our simple service. We'll go through the rest of the application as we discuss how we implement the twelve-factor methodology in the following subsections.

### 4.1. Codebase

The first best practice of twelve-factor apps is to track it in a version control system. [Git](https://git-scm.com/) is the most popular version control system in use today and is almost ubiquitous. The principle states that **an app should be tracked in a single code repository and must not share that repository with any other apps**.

Spring Boot offers many convenient ways to bootstrap an application, including a command-line tool and [a web interface](https://start.spring.io/). Once we generate the bootstrap application, we can convert this into a git repository:

git init

This command should be run from the root of the application. The application at this stage already contains a .gitignore file which effectively restricts generated files from being version-controlled. So, we can straight away create an initial commit:

git add .

git commit -m "Adding the bootstrap of the application."

Finally, we can add a remote and push our commits to the remote if we wish to (this is not a strict requirement):

git remote add origin https://github.com/<username>/12-factor-app.git

git push -u origin master

### 4.2. Dependencies

Next, the **twelve-factor app should always explicitly declare all its dependencies**. We should do this using a dependency declaration manifest. Java has multiple dependency management tools like Maven and Gradle. We can use one of them to achieve this goal.

So, our simple application depends on a few external libraries, like a library to facilitate REST APIs and to connect to a database. Let's see how can we declaratively define them using Maven.

Maven requires us to describe a project's dependencies in an XML file, typically known as [Project Object Model](https://maven.apache.org/guides/introduction/introduction-to-the-pom.html) (POM):

<**dependencies**>

<**dependency**>

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

<**artifactId**>spring-boot-starter-web</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>com.h2database</**groupId**>

<**artifactId**>h2</**artifactId**>

<**scope**>runtime</**scope**>

</**dependency**>

</**dependencies**>

Although this looks plain and simple, these dependencies usually have other transitive dependencies. This complicates it to an extent but helps us achieve our goal. Now, our application doesn't have a direct dependency which is not explicitly described.

### 4.3. Configurations

An application typically has lots of configuration, some of which may vary between deployments while others remain the same.

In our example, we've got a persistent database. We'll need the address and credentials of the database to connect to. This is most likely to change between deployments.

**A twelve-factor app should externalize all such configurations that vary between deployments**. The recommendation here is to use environment variables for such configurations. This leads to a clean separation of config and code.

Spring provides a configuration file where we can declare such configurations and attach it to environment variables:

spring.datasource.url=jdbc:mysql://${MYSQL\_HOST}:${MYSQL\_PORT}/movies

spring.datasource.username=${MYSQL\_USER}

spring.datasource.password=${MYSQL\_PASSWORD}

Here, we've defined the database URL and credentials as configurations and have mapped the actual values to be picked from the environment variable.

On Windows, we can set the environment variable before starting the application:

set MYSQL\_HOST=localhost

set MYSQL\_PORT=3306

set MYSQL\_USER=movies

set MYSQL\_PASSWORD=password

We can use a configuration management tool like [Ansible](https://www.ansible.com/) or [Chef](https://www.chef.io/) to automate this process.

### 4.4. Backing Services

Backing services are services that the application depends on for operation. For instance a database or a message broker. **A twelve-factor app should treat all such backing services as attached resources.** What this effectively means is that it shouldn't require any code change to swap a compatible backing service. The only change should be in configurations.

In our application, we've used [MySQL](https://www.mysql.com/) as the backing service to provide persistence.

[Spring JPA](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa) makes the code quite agnostic to the actual database provider. We only need to define a repository which provides all standard operations:

@Repository

**public** **interface** **MovieRepository** **extends** **JpaRepository**<**Movie**, **Long**> {

}

As we can see, this is not dependent on MySQL directly. Spring detects the MySQL driver on the classpath and provides a MySQL-specific implementation of this interface dynamically. Moreover, it pulls other details from configurations directly.

So, if we've to change from MySQL to Oracle, all we've to do is replace the driver in our dependencies and replace the configurations.

### 4.5. Build, Release and Run

The twelve-factor methodology **strictly separates the process of converting codebase into a running application** as three distinct stages:

* Build Stage: This is where we take the codebase, perform static and dynamic checks, and then generate an executable bundle like a JAR. Using a tool like [**Maven**](https://maven.apache.org/), this is quite trivial:

mvn clean compile test package

* Release Stage: This is the stage where we take the executable bundle and combine this with the right configurations. Here, we can use [**Packer**](https://www.packer.io/) with a provisioner like **[Ansible](https://www.ansible.com/)** to create Docker images:

packer build application.json

* Run Stage: Finally, this is the stage where we run the application in a target execution environment. If we use [**Docker**](https://www.docker.com/) as the container to release our application, running the application can be simple enough:

docker run --name <container\_id> -it <image\_id>

Finally, we don't necessarily have to perform these stages manually. This is where [Jenkins](https://jenkins.io/) comes in as pretty handy with their declarative pipeline.

### 4.6. Processes

**A twelve-factor app is expected to run in an execution environment as stateless processes.** In other words, they can not store persistent state locally between requests. They may generate persistent data which is required to be stored in one or more stateful backing services.

In the case of our example, we've got multiple endpoints exposed. A request on any of these endpoints is entirely independent of any request made before it. For instance, if we keep track of user requests in-memory and use that information to serve future requests, it violates a twelve-factor app.

Hence, a twelve-factor app imposes no such restriction like sticky sessions. This makes such an app highly portable and scalable. In a cloud execution environment offering automated scaling, it's quite a desirable behavior from applications.

### 4.7. Port Binding

A traditional web application in Java is developed as a WAR or web archive. This is typically a collection of Servlets with dependencies, and it expects a conformant container runtime like Tomcat.**A twelve-factor app, on the contrary, expects no such runtime dependency.** It's completely self-contained and only requires an execution runtime like Java.

In our case, we've developed an application using Spring Boot. Spring Boot, apart from many other benefits, provides us with a default embedded application server. Hence, the JAR we generated earlier using Maven is fully capable of executing in any environment just by having a compatible Java runtime:

java -jar application.jar

Here, our simple application exposes its endpoints over an HTTP binding to a specific port like 8080. Upon starting the application as we did above, it should be possible to access the exported services like HTTP.

An application may export multiple services like FTP or [WebSocket](https://www.baeldung.com/websockets-spring) by binding to multiple ports.

### 4.8. Concurrency

Java offers Thread as a classical model to handle concurrency in an application. Threads are like lightweight processes and represent multiple paths of execution in a program. Threads are powerful but have limitations in terms of how much it can help an application scale.

**The twelve-factor methodology suggests apps to rely on processes for scaling.** What this effectively means is that applications should be designed to distribute workload across multiple processes. Individual processes are, however, free to leverage a concurrency model like Thread internally.

A Java application, when launched gets a single process which is bound to the underlying JVM. What we effectively need is a way to launch multiple instances of the application with intelligent load distribution between them. Since we've already packaged our application as a [Docker](https://www.baeldung.com/docker-java-api) container, [Kubernetes](https://www.baeldung.com/kubernetes) is a natural choice for such orchestration.

### 4.9. Disposability

Application processes can be shut down on purpose or through an unexpected event. In either case, **a twelve-factor app is supposed to handle it gracefully**. In other words, an application process should be completely disposable without any unwanted side-effects. Moreover, processes should start quickly

For instance, in our application, one of the endpoints is to create a new database record for a movie. Now, an application handling such a request may crash unexpectedly. This should, however, not impact the state of the application. When a client sends the same request again, it shouldn't result in duplicate records.

In summary, the application should expose [idempotent](https://www.baeldung.com/cs/idempotent-operations) services. This is another very desirable attribute of a service destined for cloud deployments. This gives the flexibility to stop, move, or spin new services at any time without any other considerations.

### 4.10. Dev/Prod Parity

It's typical for applications to be developed on local machines, tested on some other environments and finally deployed to production. It's often the case where these environments are different. For instance, the development team works on Windows machines whereas production deployment happens on Linux machines.

**The twelve-factor methodology suggests keeping the gap between development and production environment as minimal as possible.** These gaps can result from long development cycles, different teams involved, or different technology stack in use.

Now, technology like Spring Boot and Docker automatically bridge this gap to a great extent. A containerized application is expected to behave the same, no matter where we run it. We must use the same backing services – like the database – as well.

Moreover, we should have the right processes like continuous integration and delivery to facilitate bridging this gap further.

### 4.11. Logs

Logs are essential data that an application generates during its lifetime. They provide invaluable insights into the working of the application. Typically an application can generate logs at multiple levels with varying details and output ii in multiple different formats.

A twelve-factor app, however, separates itself from log generation and its processing. **For such an app, logs are nothing but a time-ordered stream of events.** It merely writes these events to the standard output of the execution environment. The capture, storage, curation, and archival of such stream should be handled by the execution environment.

There are quite several tools available to us for this purpose. To begin with, we can use [SLF4J](https://www.baeldung.com/slf4j-with-log4j2-logback) to handle logging abstractly within our application. Moreover, we can use a tool like [Fluentd](https://www.fluentd.org/) to collect the stream of logs from applications and backing services.

This we can feed into [Elasticsearch](https://www.elastic.co/) for storage and indexing. Finally, we can generate meaningful dashboards for visualization in [Kibana](https://www.elastic.co/products/kibana).

### 4.12. Admin Processes

Often we need to perform some one-off tasks or routine procedure with our application state. For instance, fixing bad records. Now, there are various ways in which we can achieve this. Since we may not often require it, we can write a small script to run it separately from another environment.

Now, **the twelve-factor methodology strongly suggests keeping such admin scripts together with the application codebase**. In doing so, it should follow the same principles as we apply to the main application codebase. It's also advisable to use a built-in REPL tool of the execution environment to run such scripts on production servers.

In our example, how do we seed our application with the already watched movies so far? While we can use our sweet little endpoint, but that may seem to be impractical. What we need is a script to perform a one-time load. We can write a small Java function to read a list of movies from a file and save them in batch into the database.

Moreover, we can use [Groovy integrated with Java](https://www.baeldung.com/groovy-java-applications) runtime to start such processes.

## 5. Practical Applications

So, now we've seen all the factors suggested by the twelve-factor methodology. Developing an application to be a **twelve-factor app certainly has its benefits, especially when we wish to deploy them as services on the cloud**. But, like all other guidelines, framework, patterns, we must ask, is this a silver bullet?

Honestly, no single methodology in software design and development claim to be a silver bullet. The twelve-factor methodology is no exception. While **some of these factors are quite intuitive**, and most likely we're already doing them, **others may not apply to us**. It's essential to evaluate these factors in the backdrop of our objectives and then choose wisely.

It's important to note that all these factors are **there to help us develop an application which is modular, independent, portable, scalable, and observable**. Depending upon the application, we may be able to achieve them through other means better. It's also not necessary to adopt all the factors together, adopting even some of these may make us better than we were.

Finally, these factors are quite simple and elegant. They hold greater importance in an age where we demand our applications to have higher throughput and lower latency with virtually no downtime and failure. **Adopting these factors gives us the right start from the beginning.** Combined with microservice architecture and containerization of applications, they just seem to hit the right spot.

## 6. Conclusion

In this tutorial, we went through the concepts of twelve-factor methodology. We discussed how to leverage a microservice architecture with Spring Boot to deliver them effectively. Further, we explored each factor in detail and how to apply them to our application. We also explored several tools to apply these individual factors in an effective manner successfully.

# What are Advantages of Microservices Systems?

1. **Complexity localization.** One of the wonderful things about a microservice-based architecture is that it allows you to think about services as self-contained, independent applications. The development team for each service is only concerned with understanding the complexities of their service. Other teams only need to know what capabilities are being provided by the other services; they don’t need to know how they work internally. This localization of complexity and compartmentalization of knowledge can help you create and manage large applications more effectively.
2. **Cross-cutting business functionality.** The use of microservices eliminates the need to build standard pieces of functionality used across the organization multiple times; for example, authentication and user management. By developing these business services that support multiple different applications as microservices, you can break up monolithic applications into several smaller and more maintainable applications.
3. **Increased Resiliency.** Since applications consist of a number of services communicating together when one fails the client should be designed to allow its neighboring services to continue functioning as it steps out as elegantly as possible. This type of improved fault isolation means larger applications remain largely unaffected by the failure of a single module resulting in improved service availability and an uninterrupted user experience.
4. **Better scaling.** Using microservices reduces the amount of effort required to identify slow bottlenecks in an application. It also allows for individual microservices to be scaled in order to resolve those bottlenecks providing for a better overall user experience.
5. **Output Flexibility.** By simplifying the way data can be extracted for various end users, microservices allows developers to tailor the presentation of the data for different audiences easily.
6. **Real-time processing support.** The publish-subscribe framework at the core of a microservices architecture enables data processing in real time to deliver direct output and insights.
7. **Support for best technology use.** By breaking a large application’s functions up into microservices, you are no longer limited to select a single technology set for the overall project. Each service can be developed using the best programming language and data storage technology for its function.
8. **Efficient system optimization and organization**. Scaling decisions can be made at a more granular level with a microservice architecture. You can select where to apply resources to meet your scaling needs based on smaller, more granular components, allowing for more efficient system optimization and organization.
9. **Rapid growth facilitation.** Due to their modular architecture, microservices enable a high level of code and data reuse, making it faster and easier to deploy additional data-driven use cases and solutions for additional business value.
10. **Cross-functional teams.** A microservice architecture is typically organized around business capabilities and priorities. Such services take a broad-stack implementation of software including project management, user-experience, persistent storage, and any external collaborations so they can deliver services communicating via message bus; consequently, the teams are cross-functional.
11. **Outsourcing flexibility.** While many business owners want to be able to offload work to third-party partners, they are often concerned about protecting their intellectual property. Microservices allows the business to segment off work for outsourcing of their non-core business functions without disclosing their core services.
12. **Team optimization.** Due to their focused functionality, microservices allow you to create optimized teams by selecting members based on their specific capabilities allowing them to fully focus on the particular scaling and availability requirements of their assigned service(s).
13. **Technology experimentation flexibility.** Since each microservice is technically independent, you have the flexibility to try out a new technology stack on an individual service. Compared to monolithic designs, any dependency concerns you might have will be smaller and rolling back changes is simpler since there is less code in play. As a resulting bonus, they inherently eliminate the classic long-term commitment to a single technology stack most companies are all too familiar with.
14. **Cross-team coordination support.** Microservices use event-streaming technologies to enable easy integration compared to the heavyweight inter-process communications protocols of traditional SOA architectures.
15. **High-quality.** Breaking a large application into components by business capability enables development teams to focus on one small set of functionalities at a time. The overall coding and testing process for individual services is more precise as a result.

# Disadvantages of Microservices

Now let’s take a look at the negative side of microservices and explore their possible disadvantages. Many of the disadvantages are similar to those with implementing any new technology or architecture. Just as with the benefits, the specific disadvantages that might apply to your project are directly related to its specifics, your team, and your organization. Here are the main disadvantages you need to be aware of as you look into utilizing microservices:

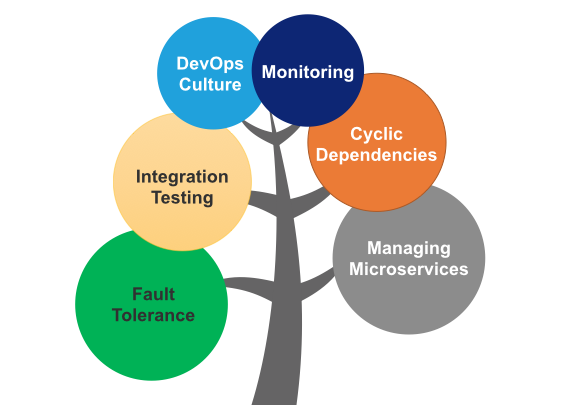
1. There is not a commonly agreed upon definition of the size, scope, and boundaries of a microservice.
2. Implementing microservices requires a change in development mindset: the architecture should be designed with managing service failure as a top-level requirement.
3. A high degree of complexity exists at a higher level regarding orchestrating the services and business processes throughout them.
4. Maintaining the complex and rich business logic in enterprise applications can be difficult to maintain in a microservice architecture.
5. A high level of communication between teams is required to make sure an update to one service doesn’t break some other functionality.
6. Teams must maintain updated schemas and interface documents to help other teams using their service resulting in increased documentation overhead.
7. Organizational changes may be required to give teams the power they need to achieve the necessary level of independence they must have to cover the whole lifecycle of a microservice.
8. Microservice increase the overall operational complexity of a project and as a result, require a mature operations team to manage the services.
9. Testing a microservices-based application with its asynchronous communication and distributed nature can be challenging.
   1. Each dependent service needs to be confirmed before you can test the application as a whole.
   2. Understanding, managing and testing dependencies is challenging.
   3. Automation testing gets more difficult when each microservices is running on a different runtime environment.
10. Large numbers of microservices are harder to secure.
11. The issues with distributed systems are introduced into the application including development, deployment, and operational management overheads.
    1. Time to market may be impacted.
    2. Remote calls may experience latency and are at risk of failure.
    3. Maintaining high levels of consistency is difficult.
12. Writing extra code to handle service issues is required to avoid application disruption.
13. Deploying microservices can be complex and result in increased operational costs when services span across multiple systems since different service types require additional deployment configuration files and job scripts to run.
14. There are increased configuration management needs across the application since each microservice requires a dedicated build and delivery pipeline.
15. Implementing transactions correctly and maintaining their safety can be challenging when multiple microservices are involved.
16. Operational expenditures and overhead could be significantly higher depending on the complexity of the system.
    1. There is a higher initial investment required to run these applications as all the independently running components need their own runtime containers and utilize more memory and CPU.
    2. There is a larger overall memory footprint resulting in increased memory consumption since you are replacing the number of monolithic application instances with an increased number of services instances and potentially additional instances of some utility classes and libraries.
    3. Reliable and fast network connections are required to support the increased amount of network communication from services interacting with each other.
    4. Additional processing power is required as data is moved between services.
17. The costs and complexity involved in monitoring applications using microservices in production are higher.
18. Applications may see a decrease in performance since inter-process communication is slower than shared process communication and message flow increases with the number of microservices.
19. Release risk is increased since even simple cross-cutting changes require changes to many other components, all having to be released in coordinated ways.
20. A change in requirements requires you that you either create a new microservice for the new feature or rewrite and add it to an existing service.
21. If you need to moving code between services, you must change all dependent services making refactoring hard.
22. Maintaining dependent services compatibility can be challenging when updating a single service even when using versioning. Plus, there is no well-defined way to handle versioning of services.
23. Being able to choose a different technology stack for various services results in non-uniform application design and architecture which can increase long-term maintenance costs.

By being fully aware of these negative points, you will be better able to make an educated decision about whether or not utilizing microservices is appropriate in your case and have the opportunity to address them should you decide to [move forward with microservices](https://www.qat.com/microservices/).

# Challenges in Implementing Microservices

We live in a world of microservices. "Monolith to microservices" is a phrase we hear from more than 70% of technology leaders today. The benefits are well-documented: increased resilience, improved scalability, faster time to market. Like most transformational trends, implementing microservices poses its own challenges. It is imperative that these challenges are well understood, or be prepared for your project to never see the light of day, or finish it to only see that many of the foreseen benefits are not being achieved.

Here are some of the top challenges that organizations face in their microservices journey:



## **Managing Microservices**

As the number of microservices increases, managing them gets more challenging. It is important that management is planned before or while microservices are being built. While the modularity helps, things can very quickly get out of hand if not managed well. Many engineering leaders have stated that the mismanagement of these services is as much a problem as problems faced during the initial stages of the transformation from monolithic applications.

Developing microservices management tooling on your own, although a valid option, can be complex and cumbersome. We recommend looking to acquire a platform whose capabilities include microservices management.

## **Monitoring**

The traditional forms of monitoring and diagnostics will not align well with microservices since you have multiple services making up the same functionality previously supported by a single application. When a problem arises in the application, finding the root cause can be challenging if you do not have a means of monitoring and tracking the path a specific request took, like how many and which microservices were traversed for a specific request coming from a user interface.

We have seen customers who struggle to analyze the chain of communication across these services and where issues were potentially introduced. [This](https://www.youtube.com/watch?v=smEuX-Hq6RI) video is a good watch on this topic.

## **Embracing DevOps Culture**

Separate teams need agility, autonomy, and continuous delivery to be able to deliver initial releases and subsequent iterative changes. A lack of DevOps culture can bottle up releases and impact the overall time to market and the response to business requests and issues.

## **Fault Tolerance**

It is important that individual services do not bring down the overall system. Fault tolerance at the service level, and more importantly, at the overall solution level, is critical. Given the complexity of a microservices environment and the complex dependency chains, failure is inevitable. Microservices need to be able to withstand both internal and external failures. Robust resiliency testing is key to successful issue preparedness.

## **Testing**

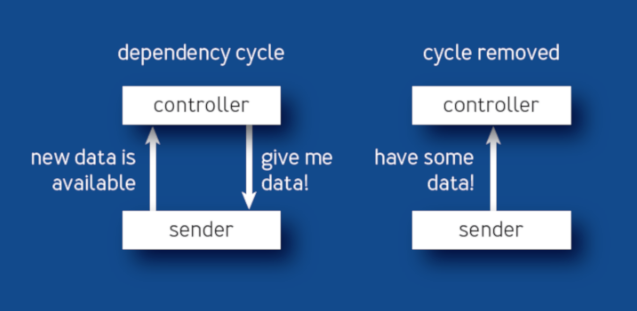
Testing is much more complex in a microservices environment due to the different services, their integration, and interdependencies. The team members responsible for quality assurance need to be knowledgeable on the order and channels of communications between services to have full coverage in their test cases. The asynchronous aspect of microservices also makes it harder to test in lower environments. Indistinct behaviors from microservices are harder to predict and validate.

More details on microservice testing challenges can be found [here](http://www.alohatechnology.com/blog/testing-challenges-in-a-microservices-environment.html) and in [this insightful post](https://bit.ly/2meWzaF).

## Design With Failure in Mind

While this is counter-intuitive to many, expecting failure scenarios and building a robust set of microservices is imperative to a successful implementation. When more failure situations are predicted during design, the more exception handling mechanisms will be built and seamless resolution of issues will be handled better. This is easier said than done.

## **Cyclic Dependencies**



Source: [*queue.acm.org*](http://queue.acm.org/).

Dependency management across different services and their functionality is very important and cyclic dependencies can be a headache if not identified and resolved promptly. In microservice architecture, you’re even more vulnerable to errors coming from dependency issues. Decisions made around upgrades on related services with these dependencies are critical. [This post](https://queue.acm.org/detail.cfm?id=3277541) discusses tracking and controlling dependencies in detail.

# Challenges of Microservices Architecture

Microservice architecture is more complex than the legacy system. The microservice environment becomes more complicated because the team has to manage and support many moving parts. Here are some of the top challenges that an organization face in their microservices journey:

* Bounded Context
* Dynamic Scale up and Scale Down
* Monitoring
* Fault Tolerance
* Cyclic dependencies
* DevOps Culture

**Bounded context**: The bounded context concept originated in Domain-Driven Design (DDD) circles. It promotes the Object model first approach to service, defining a data model that service is responsible for and is bound to. A bounded context clarifies, encapsulates, and defines the specific responsibility to the model. It ensures that the domain will not be distracted from the outside. Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.

In other words, the service owns its data and is responsible for its integrity and mutability. It supports the most important feature of microservices, which is independence and decoupling.

**Dynamic scale up and scale down**: The loads on the different microservices may be at a different instance of the type. As well as auto-scaling up your microservice should auto-scale down. It reduces the cost of the microservices. We can distribute the load dynamically.

**Monitoring**: The traditional way of monitoring will not align well with microservices because we have multiple services making up the same functionality previously supported by a single application. When an error arises in the application, finding the root cause can be challenging.

**Fault Tolerance**: Fault tolerance is the individual service that does not bring down the overall system. The application can operate at a certain degree of satisfaction when the failure occurs. Without fault tolerance, a single failure in the system may cause a total breakdown. The circuit breaker can achieve fault tolerance. The circuit breaker is a pattern that wraps the request to external service and detects when they are faulty. Microservices need to tolerate both internal and external failure.

**Cyclic Dependency**: Dependency management across different services, and its functionality is very important. The cyclic dependency can create a problem, if not identified and resolved promptly.

**DevOps Culture**: Microservices fits perfectly into the DevOps. It provides faster delivery service, visibility across data, and cost-effective data. It can extend their use of containerization switch from Service-Oriented-Architecture (SOA) to Microservice Architecture (MSA).

## **Other challenges of microservices**

* As we add more microservices, we have to be sure they can scale together. More granularity means more moving parts, which increase complexity.
* The traditional logging is ineffective because microservices are stateless, distributed, and independent. The logging must be able to correlate events across several platforms.
* When more services interact with each other, the possibility of failure also increases.

# Migrating from Monolith to Microservices

Scalability is one of the primary motivations for moving to a microservice architecture. Moreover, the scalability effect on rarely used components is negligible. Components that are used by the most users should therefore be considered for migration first.

Users want their interactions with a system to return the right data at the right level of detail, usually as fast as that data can be acquired. The jobs for users each involve one or more data objects, and each data object has a set of associated actions that can be performed. The development team that designs and implements the system must consider the collection of jobs, data objects, and data actions. A typical process to migrate from a monolithic system to a microservices-based system involves the following steps:

1. Identify logical components.
2. Flatten and refactor components.
3. Identify component dependencies.
4. Identify component groups.
5. Create an API for remote user interface.
6. Migrate component groups to macroservices (move component groups to separate projects and make separate deployments).
7. Migrate macroservices to microservices.
8. Repeat steps 6-7 until complete.

I describe "components," the "remote user interface," and "macroservices" in more detail below.

In this process, developers perform steps 1-5 once at the beginning of the migration project. The team can revisit these steps as processes get migrated or when new information becomes available during the migration process, but once the steps are completed, it should not be necessary to revisit them.

Step 6 is an interim step that may not be needed for some projects. If a macroservice is not needed, developers can skip step 6; step 7 then becomes, "Migrate component groups to microservices." Likewise, developers can skip step 7 or a particular iteration if the migration from macroservice to microservice appears too hard or if the macroservice transition is adequate by itself.

Step 8 exists because developers need not perform the entire migration at once. They can pull components out of the legacy monolith and make them into microservices, either singly or in groups as the development warrants.

**1.** **Identify Logical Components**

There are three main information components with the data used in the system:

* data objects
* data actions
* job to perform and use cases

The data objects are the logical constructs representing the data being used. The data actions are the commands that are used on one or more data objects, possibly on different types of data, to perform a task. The job to perform represents the function the users are calling to fulfill their organizational roles. The jobs to perform may be captured as [use cases](https://en.wikipedia.org/wiki/Use_case), [user stories](https://en.wikipedia.org/wiki/User_story), or other documentation involving user input.

When combining multiple systems into a unified system, the data objects, data actions, and jobs to perform for each individual system must be identified. All these components are implemented as modules within the codebase with one or more modules representing each data object, data action, and job to perform. These modules should be grouped into categories for working with later steps. This grouping is indicated by color coding in Figure 1.

System architects may find it easiest to identify the data objects used within a system. Working from this dataset, they can then determine the data actions and map these to the jobs to performed by users of the system. The codebase is usually [object-centric](https://stackoverflow.com/questions/9346995/what-is-the-difference-between-data-centric-and-object-oriented-application-mode), and each code object is associated with functions and jobs to perform.

During this part of the migration process, system architects should be asking the following questions:

* If two or more applications provide similar data, can this data be merged?
* What should be done about data fields being different or missing in similar objects?

The migration from a monolithic system to microservices does not typically affect the user interface directly. The components that are best for migrating are thus determined by which components

* are used by the most users
* are used most frequently
* have the fewest dependencies on other components
* are performing too slowly

**2. Flatten and Refactor Components**

After all the modules have been uniquely identified and grouped, it is time to organize the groups internally. Components that duplicate functionality must be addressed before implementing the microservice. In the final system, there should be only one microservice that performs any specific function. Function duplication will most likely be encountered when there are multiple monolithic applications being merged. It may also arise where there is legacy (possibly dead) code that is included in a single application.

Merging duplicated functions and data will require the same considerations as when designing the ingestion of a new dataset:

* Check data formats.
* Verify datatypes.
* Verify data accuracy.
* Verify data units.
* Identify outliers.
* Deal with missing fields or values.

Since one of the effects of this migration is to have a single data repository for any piece of data, any data that is replicated in multiple locations must be examined here, and the final representation must be determined. The same data may be represented differently depending on the job to be done. It is also possible that similar data may be obtained from multiple locations, or that the data may be a combination from multiple data sources. Whatever the source and however the data will be used, it is essential that one final representation exists for each unique datatype.

**3. Identify Component Dependencies**

After the components have been identified and reorganized to prepare for the migration, the system architect should identify the dependencies between the components. This activity can be performed using a [static analysis](https://en.wikipedia.org/wiki/Static_program_analysis) of the source code to search for calls between different libraries and datatypes. There are also several [dynamic-analysis](https://en.wikipedia.org/wiki/Dynamic_program_analysis) tools that can analyze the usage patterns of an application during its execution to provide an automated map between components. Figure 2 below shows an example of a map of component dependencies.

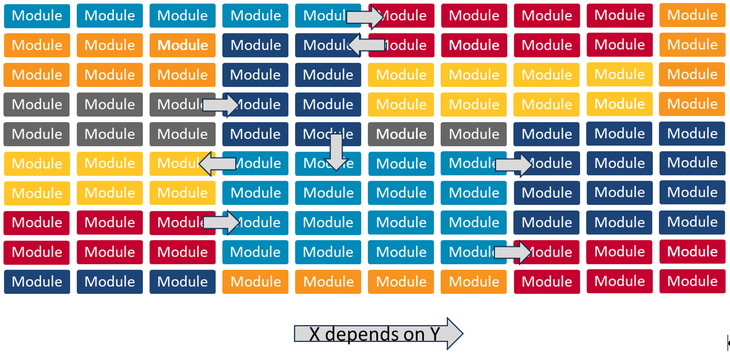
[](https://insights.sei.cmu.edu/media/images/Microservices_fig_2.1.original.png)

Figure 2: Identify Component Dependencies

One tool that can be used for identifying component dependencies is [SonarGraph-Explorer](https://www.hello2morrow.com/products/sonargraph/explorer). This tool includes a view of the elements arrayed in a circle or in a hierarchy, which allows an analyst to visualize how each component is associated with other components in the codebase.

**4. Identify Component Groups**

After the dependencies have been identified, the system architect should focus on grouping the components into cohesive groups that can be transformed into microservices, or, at least, macroservices. The distinction between macroservices and microservices at this point is not important. The goal is to identify a small set of objects and their constituent actions that should be logically separated in the final system.

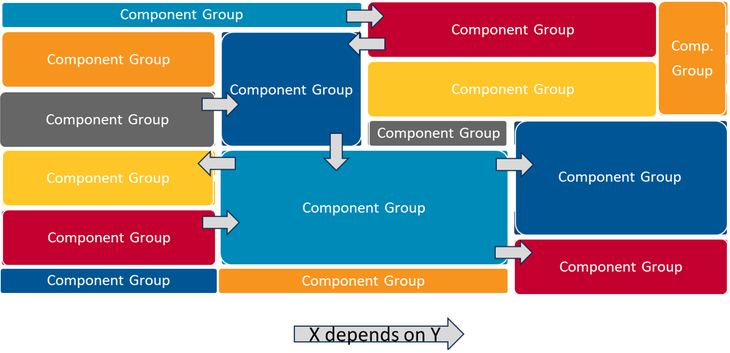
[](https://insights.sei.cmu.edu/media/images/Microservices_fig_3_ueefbE1.original.png)

Figure 3: Identify Component Groups

**5. Create an API for Remote User Interface**

The remote user interface is intended as the sole mode of communication between the system, its components, and the system's users. It is vitally important that this interface be scalable and well planned to avoid problems as the system evolves over time. The underlying interface must be usable both during the migration and afterwards, so it is likely change as components are reworked from the monolithic system to macroservices and microservices.

The key output from this migration effort is a unified API that the user interface(s) and applications can use to manipulate the data. Everything else depends on this API, so it should be engineered to ensure that existing data interactions will not change significantly. Instead, it should allow for the addition of new objects, attributes, and actions as they are identified and made available. After the API layer is in place, all new functionality should be added through the API, not through the legacy applications.

The design and implementation of the API is key to the success of the migration to microservices. The API must be able to handle all data-access cases supported by the applications that will use the API. In keeping with standard practice of [software version-control](https://en.wikipedia.org/wiki/Version_control) numberings, a change to the API that breaks the backward compatibility with earlier versions should trigger a change of the "major" revision number for the API.

Changes to the API that break backward compatibility should be infrequent and planned in advance to prevent recurring deployment problems. The API should provide a mechanism so that the application can check the API version being used and warn users and developers about incompatibilities. The only changes to the API should be those that add new data objects and functions and that do not modify the format of the existing outputs or expected inputs. For microservices to work properly, all data access must be provided through the API to the micro-services or, during the migration transition period, to the macroservices or legacy application.

To maximize the scalability of the final system, the API should be

* stateless
* able to handle all data objects represented within the system
* backward compatible with previous versions
* versioned

Although a [REST API](https://en.wikipedia.org/wiki/Representational_state_transfer) is typical, it is not strictly mandated for microservices.

**6. Migrate Component Groups to Macroservices**

Macroservices have a more relaxed posture toward sharing data repositories and allow more complex interactions with data objects. It may therefore be useful to consider their use as an interim step toward migrating to full microservices. The main reason for not moving directly to microservices is complexity. A monolithic system is typically built with intertwined logic that may cause problems when converting to microservices. If the monolith is continuously changing, then migrating to microservices in a single step will be a continuously changing target as well.

The key goal at this step is to move component groups into separate projects and make separate deployments. At a minimum, each macroservice should be independently deployable from within the system's [continuous integration (CI)](https://insights.sei.cmu.edu/blog/continuous-integration-in-devops/) and [continuous deployment (CD)](https://www.youtube.com/watch?v=Bx7GcZm7fCE) pipeline.

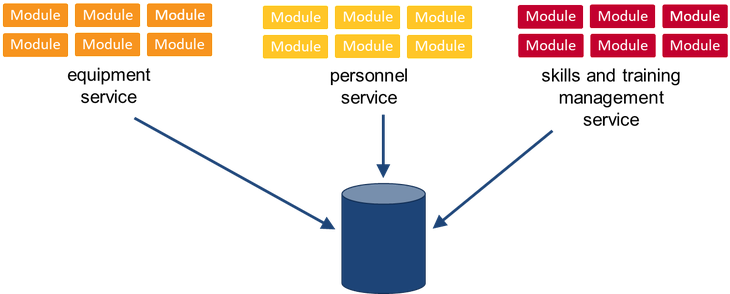
[](https://insights.sei.cmu.edu/media/images/Microservices_fig_4.original.png)

Figure 4: Migrate Component Groups to Macroservices

**7. Migrate Macroservices to Microservices**

The process of pulling the components, data objects, and functions out of the monolithic system and into macroservices will provide insight into how these components can be further separated into microservices. Remember, each microservice maintains its own datastore and performs only a small set of actions on the data objects within that datastore.

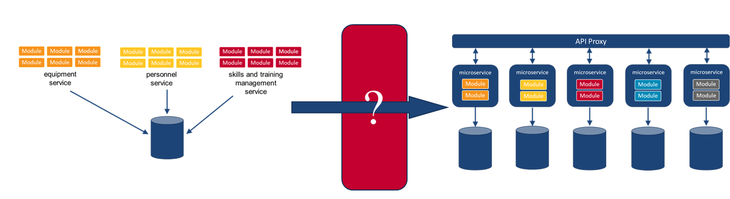
[](https://insights.sei.cmu.edu/media/images/Microservices_fig_5.original.png)

Figure 5: Migrate from Macroservice to Microservices

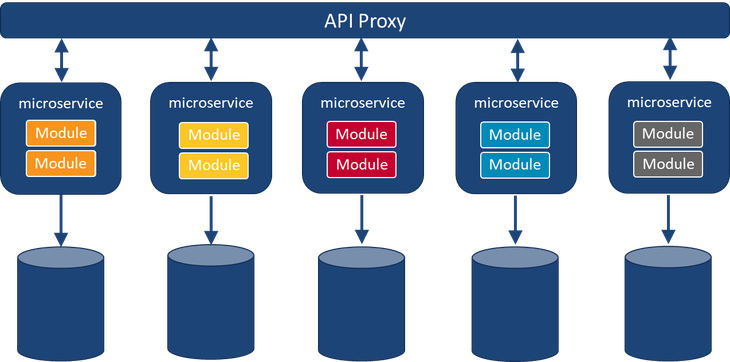
[](https://insights.sei.cmu.edu/media/images/Microservices_fig_6_varVjdj.original.png)

Figure 6: Final Result: Application Fully Converted to Microservices

**8. Deployment and Testing**

Once a macroservice or microservice is ready for deployment, the next step involves [integration testing](https://en.wikipedia.org/wiki/Integration_testing) and deployment. The monolithic system must be configured to use the new service for its data needs rather than its legacy datastore.

Finding all calls to the datastore from within the legacy monolithic system can be hard. In a testing environment, it should be possible to remove the legacy data related to the migrated dataset for which the new microservice is now responsible. All functions that access the migrated data should be tested in all user interfaces to ensure that there is no function that still attempts to use the old datastore through a previously undetected method. If possible, accesses to the old dataset on the old datastore should be logged and flagged in case old or refactored code is still able to access the legacy data. Access controls should be updated to prevent users from accessing the old data directly from the datastore; users may be notified how to access the data using the new API interface if such direct accesses are permitted.

After testing indicates that the remaining monolithic code is accessing the new service for its information and that there don't appear to be any remaining connections with the old datastore, the new service can be deployed to the production systems.

**Consolidating Multiple Services**

When this process is used to consolidate multiple services rather than just migrating a single legacy system, there may be one or more instances where the same or similar data is stored by more than one legacy system. This consolidation poses a problem of how to handle the data during the migration period. At the end of the migration, this data should be stored as one or more microservices and accessed through the API by all components that want to use that data.

Ideally, as part of Step 2, Flatten and Refactor Components, a single authoritative system should be identified to handle the data and all dataflows would be configured to use this single system. The API would point to this system, and the legacy applications would be configured to use the API to access this data in its authoritative location.

The API should always read data from one authoritative location. Ideally, this data should always write data to a single authoritative location as well. However, an interim macroservice may be configured to write data to multiple datastores concurrently, possibly including the datastore for the legacy monolithic system. At some point before the completion of the migration process, the multiple *writes* must be removed so that only a single datastore contains the relevant data.

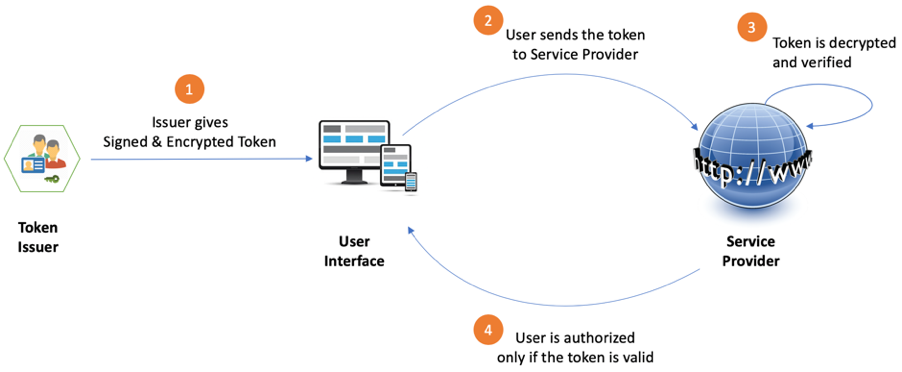
**Wrapping Up**

This plan for migrating existing applications to microservices is intended to enable organizations to realize benefits of microservice architectures, such as resilience, scalability, improved time to market, and easier maintenance, with maximum efficiency and minimal disruption to existing applications and services.

# How to Authenticaton systems microservices using JWT?

## **Technical Overview of JWT-Based Authorization**

Like the real-world analogy explained above, JWT authorization involves three entities and four steps, as described in the sequence diagram below:



### **Entities in JWT Authorization Flow**

1. **Token Issuer**–The issuer of the token (Identity and Access management system)
2. **User Interface**–The user’s browser or mobile app
3. **Service Provider**– The web site or app that the user wants to access

### **Steps in JWT Authorization**

#### **Step 1: Token Issuer Gives a Signed & Encrypted Token to User Interface**

The user authenticates to Token Issuer using some login method and asks the Token Issuer to grant a token. Upon success authentication, the Token Issuer creates a JSON Web token (JWT) which has the following structure:

**Header.Payload. Signature**

### Anatomy of a JWT

It consists of three parts

* **Header**: Consists of info like signature mechanism, token type etc.
* **Body (Claims)**: The meat of the payload.
* **Signature**: Signature of the contents to protect against tampered/malicious JWTs.

These three components come together to form the actual token:

//header

{

"alg": "HS256",

"typ": "JWT"

}

//payload/claims

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

//the formula

encoded\_part = base64Of(header) + "." base64Of(payload)

signature = signedUsingHS256WithSecret(encoded\_part) //assume that algo is HS256 and secret key is 'secret'

JWT = encoded\_part + "." + sigature

//the JWT ( notice the separator/period --> "." )

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 //base-64 encoded header

.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9 //base-64 encoded payload

.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ //the signature

### Benefits

* Useful for implementing Stateless authentication.
* Compact: less verbose compared to other counterparts like SAML).
* Flexible: Although its backed by a standard, you are free to choose your signature, claim attributes etc.

### Please Note That:

* JWT is not only an authentication mechanism. It’s more about information exchange, and it’s usage is limited by your imagination.
* It’s signed, notencrypted: its contents can be picked up over the wire if you do not secure your transport layer (e.g. using HTTPS).

More information on JWT structure can be found at <https://jwt.io/>

The payload part of the token contains key-value pairs called claims that provide information on who the user is and what the user is allowed to access. The Token Issuer creates a token with some claims and sends the token to user. Once the token is created and given to the user, the responsibility of using the token to gain access to service lies entirely with the user. The user is required to submit the token to the service provider, in order to obtain access.

As the responsibility of carrying the token is being given to the user, the following security concerns must be addressed:

* **Concern #1:** What if the sensitive data in the token gets tapped by an eavesdropper?
* **Concern #2:** What if the token is tampered to request services meant for other users?

The well-known RSA public key cryptography comes in handy to tackle these issues. The Token Issuer applies signature and encryption as follows:

1. Token Issuer signs the token with its private key and creates JWS (JWT – Signed).
2. Token Issuer then encrypts the JWS with the public key of the Service Provider. The ncrypted JWS is called JWE (JWT – encrypted).

Only the signed and encrypted token (JWE) is passed over to the user interface.

The user can only carry the JWE but cannot decrypt it. Only the Service Provider can decrypt the token and see the claims contained in it. Any tampering of token will also be detected by the Service Provider, as the token is signed by Token Issuer.

JWE is embedded as an Authorization header in the HTTP response sent to the client.

The authorization header appears as follows:

1

Authorization: Bearer encrypted-json-web-token-text

#### **Step 2: User Interface Sends Token Along With Request to Service Provider**

The user interface attaches the JWE as an Authorization Header to the HTTP request that it submits to the Service Provider.

#### **Step 3: Service Provider Validates the Token**

On receiving a request from user, the Service provider performs the following sequence of validations:

1. Does the request have a token?
2. Can the token be decrypted?
3. Was the token content tampered with?
4. Is the token valid?
5. What are the claims inside the token?

At the end of validation, the Service Provider extracts the payload from the JWT and finds out the claims.

Here is an example of a JWT payload that the Service provider extracts

**Claims found in the above payload:**

|  |  |
| --- | --- |
| **Claim** | **Description** |
| **iss** | **The issuer of token (Token Issuer)** |
| **aud** | **Audience (The Service Provider that the token is meant for)** |
| **iat** | **Issued-at (Time at which the token was issued)** |
| **exp** | **Expiry (Time at which the token expires)** |
| **jti** | **JWT Token ID (A unique ID or randomly generated nonce)** |
| **username** | **User name** |
| **account** | **Account number of the user** |

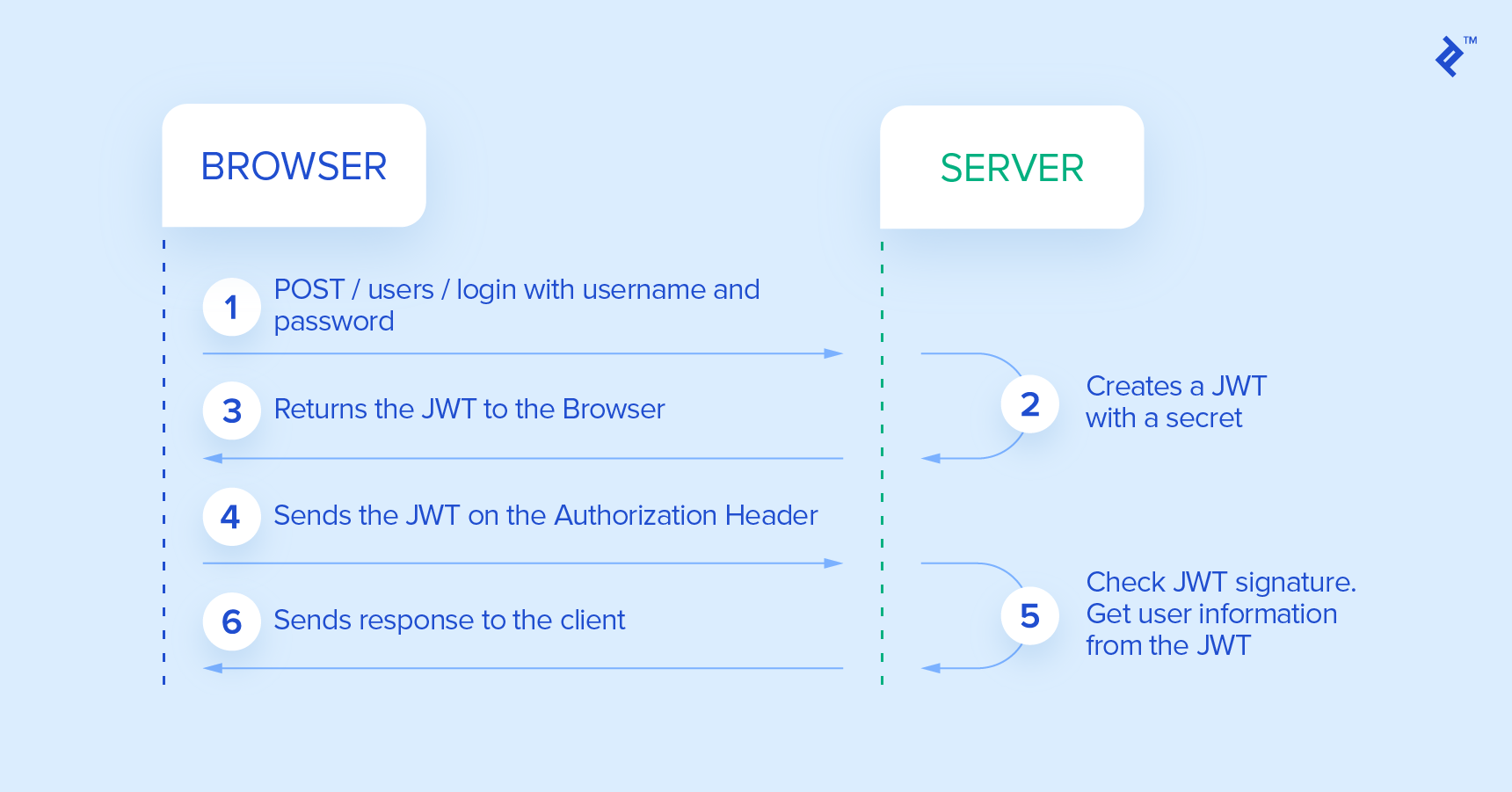
#### **Step 4: Service Provider Responds to User Interface**

Service Provider gives the appropriate response to the user interface, based on the token and the claims.

## **JWT Authentication**

JWT (shortened from JSON Web Token) is the missing standardization for using tokens to authenticate on the web in general, not only for REST services. Currently, it is in draft status as [RFC 7519](https://tools.ietf.org/html/rfc7519). It is robust and can carry a lot of information, but is still simple to use even though its size is relatively small. Like any other token, JWT can be used to pass the identity of authenticated users between an identity provider and a service provider (which are not necessarily the same systems). It can also carry all the user’s claim, such as authorization data, so the service provider does not need to go into the database or external systems to verify user roles and permissions for each request; that data is extracted from the token.

Here is how JWT security is designed to work:



* Clients logs in by sending their credentials to the identity provider.
* The identity provider verifies the credentials; if all is OK, it retrieves the user data, generates a JWT containing user details and permissions that will be used to access the services, and it also sets the expiration on the JWT (which might be unlimited).
* Identity provider signs, and if needed, encrypts the JWT and sends it to the client as a response to the initial request with credentials.
* Client stores the JWT for a limited or unlimited amount of time, depending on the expiration set by the identity provider.
* Client sends the stored JWT in an Authorization header for every request to the service provider.
* For each request, the service provider takes the JWT from the Authorization header and decrypts it, if needed, validates the signature, and if everything is OK, extracts the user data and permissions. Based on this data solely, and again without looking up further details in the database or contacting the identity provider, it can accept or deny the client request. The only requirement is that the identity and service providers have an agreement on encryption so that service can verify the signature or even decrypt which identity was encrypted.

This flow allows for great flexibility while still keeping things secure and easy to develop. By using this approach, it is easy to add new server nodes to the service provider cluster, initializing them with only the ability to verify the signature and decrypt the tokens by providing them a shared secret key. No session replication, database synchronization or inter-node communication is required. REST in its full glory.

The main difference between JWT and other arbitrary tokens is the standardization of the token’s content. Another recommended approach is to send the JWT token in the Authorization header using the Bearer scheme. The content of the header should look like this:

Authorization: Bearer <token>

## **REST Security Implementation**

For REST services to work as expected, we need a slightly different authorization approach compared to classic, multi-page websites.

Instead of triggering the authentication process by redirecting to a login page when a client requests a secured resource, the REST server authenticates all requests using the data available in the request itself, the JWT token in this case. If such an authentication fails, redirection makes no sense. The REST API simply sends an HTTP code 401 (Unauthorized) response and clients should know what to do; for example, a browser will show a dynamic div to allow the user to supply the username and password.

On the other hand, after a successful authentication in classic, multi-page websites, the user is redirected by using HTTP code 301 (Moved permanently), usually to a home page or, even better, to the page the user initially requested that triggered the authentication process. With REST, again this makes no sense. Instead we would simply continue with the execution of the request as if the resource was not secured at all, return HTTP code 200 (OK) and expected response body.

**Spring Security Example**



Now, let’s see how can we implement the JWT token based REST API using [Java](https://www.toptal.com/java) and [Spring](https://www.toptal.com/spring/beginners-guide-to-mvc-with-spring-framework), while trying to reuse the Spring Security default behavior where we can.

As expected, Spring Security framework comes with many ready to plug-in classes that deal with “old” authorization mechanisms: session cookies, HTTP Basic, and HTTP Digest. However, it lacks the native support for JWT, and we need to get our hands dirty to make it work. For a more detailed overview, you should consult official [Spring Security documentation](https://docs.spring.io/spring-security/site/docs/5.0.0.RELEASE/reference/htmlsingle/).

Now, let’s get started with the usual **Spring Security filter definition** in web.xml:

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Note that the name of the Spring Security filter must be exactly springSecurityFilterChain for the rest of the Spring config to work out of the box.

Next comes the XML declaration of the Spring beans related to security. In order to simplify the XML, we will set the default namespace to security by adding xmlns="http://www.springframework.org/schema/security" to the root XML element. The rest of the XML looks like this:

<global-method-security pre-post-annotations="enabled" /> (1)

<http pattern="/api/login" security="none"/> (2)

<http pattern="/api/signup" security="none"/>

<http pattern="/api/\*\*" entry-point-ref="restAuthenticationEntryPoint" create-session="stateless"> (3)

<csrf disabled="true"/> (4)

<custom-filter before="FORM\_LOGIN\_FILTER" ref="jwtAuthenticationFilter"/> (5)

</http>

<beans:bean id="jwtAuthenticationFilter" class="com.toptal.travelplanner.security.JwtAuthenticationFilter"> (6)

<beans:property name="authenticationManager" ref="authenticationManager" />

<beans:property name="authenticationSuccessHandler" ref="jwtAuthenticationSuccessHandler" /> (7)

</beans:bean>

<authentication-manager alias="authenticationManager">

<authentication-provider ref="jwtAuthenticationProvider" /> (8)

</authentication-manager>

* (1) In this line, we activate @PreFilter, @PreAuthorize, @PostFilter, @PostAuthorize annotations on any spring beans in the context.
* (2) We define the login and signup endpoints to skip security; even “anonymous” should be able to do these two operations.
* (3) Next, we define the filter chain applied to all requests while adding two important configs: Entry point reference and setting the session creation to stateless (we do not want the session created for security purposes as we are using tokens for each request).
* (4) We do not need csrf protection because our tokens are immune to it.
* (5) Next, we plug in our special authentication filter within the Spring’s predefined filter chain, just before the form login filter.
* (6) This bean is the declaration of our authentification filter; since it is extending Spring’s AbstractAuthenticationProcessingFilter, we need to declare it in XML to wire its properties (auto wire does not work here). We will explain later what the filter does.
* (7) The default success handler of AbstractAuthenticationProcessingFilter is not good enough for REST purposes because it redirects the user to a success page; that is why we set our own here.
* (8) The declaration of the provider created by the authenticationManager is used by our filter to authenticate users.

Now let’s see how we implement the specific classes declared in the XML above. Note that Spring will wire them for us. We start with the simplest ones.

### **RestAuthenticationEntryPoint.java**

public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint {

@Override

public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException {

// This is invoked when user tries to access a secured REST resource without supplying any credentials

// We should just send a 401 Unauthorized response because there is no 'login page' to redirect to

response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized");

}

}

As explained above, this class just returns HTTP code 401 (Unauthorized) when authentication fails, overriding default Spring’s redirecting.

### **JwtAuthenticationSuccessHandler.java**

public class JwtAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

@Override

public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) {

// We do not need to do anything extra on REST authentication success, because there is no page to redirect to

}

}

This simple override removes the default behavior of a successful authentication (redirecting to home or any other page the user requested). If you are wondering why we do not need to override the AuthenticationFailureHandler, it is because default implementation will not redirect anywhere if its redirect URL is not set, so we just avoid setting the URL, which is good enough.

### **JwtAuthenticationFilter.java**

public class JwtAuthenticationFilter extends AbstractAuthenticationProcessingFilter {

public JwtAuthenticationFilter() {

super("/\*\*");

}

@Override

protected boolean requiresAuthentication(HttpServletRequest request, HttpServletResponse response) {

return true;

}

@Override

public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {

String header = request.getHeader("Authorization");

if (header == null || !header.startsWith("Bearer ")) {

throw new JwtTokenMissingException("No JWT token found in request headers");

}

String authToken = header.substring(7);

JwtAuthenticationToken authRequest = new JwtAuthenticationToken(authToken);

return getAuthenticationManager().authenticate(authRequest);

}

@Override

protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult)

throws IOException, ServletException {

super.successfulAuthentication(request, response, chain, authResult);

// As this authentication is in HTTP header, after success we need to continue the request normally

// and return the response as if the resource was not secured at all

chain.doFilter(request, response);

}

}

This class is the entry point of our JWT authentication process; the filter extracts the JWT token from the request headers and delegates authentication to the injected AuthenticationManager. If the token is not found, an exception is thrown that stops the request from processing. We also need an override for successful authentication because the default Spring flow would stop the filter chain and proceed with a redirect. Keep in mind we need the chain to execute fully, including generating the response, as explained above.

### **JwtAuthenticationProvider.java**

public class JwtAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider {

@Autowired

private JwtUtil jwtUtil;

@Override

public boolean supports(Class<?> authentication) {

return (JwtAuthenticationToken.class.isAssignableFrom(authentication));

}

@Override

protected void additionalAuthenticationChecks(UserDetails userDetails, UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {

}

@Override

protected UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {

JwtAuthenticationToken jwtAuthenticationToken = (JwtAuthenticationToken) authentication;

String token = jwtAuthenticationToken.getToken();

User parsedUser = jwtUtil.parseToken(token);

if (parsedUser == null) {

throw new JwtTokenMalformedException("JWT token is not valid");

}

List<GrantedAuthority> authorityList = AuthorityUtils.commaSeparatedStringToAuthorityList(parsedUser.getRole());

return new AuthenticatedUser(parsedUser.getId(), parsedUser.getUsername(), token, authorityList);

}

}

In this class, we are using Spring’s default AuthenticationManager, but we inject it with our own AuthenticationProvider that does the actual authentication process. To implement this, we extend the AbstractUserDetailsAuthenticationProvider, which requires us only to return UserDetails based on the authentication request, in our case, the JWT token wrapped in the JwtAuthenticationToken class. If the token is not valid, we throw an exception. However, if it is valid and decryption by JwtUtil is successful, we extract the user details (we will see exactly how in the JwtUtil class), without accessing the database at all. All the information about the user, including his or her roles, is contained in the token itself.

### **JwtUtil.java**

public class JwtUtil {

@Value("${jwt.secret}")

private String secret;

/\*\*

\* Tries to parse specified String as a JWT token. If successful, returns User object with username, id and role prefilled (extracted from token).

\* If unsuccessful (token is invalid or not containing all required user properties), simply returns null.

\*

\* @param token the JWT token to parse

\* @return the User object extracted from specified token or null if a token is invalid.

\*/

public User parseToken(String token) {

try {

Claims body = Jwts.parser()

.setSigningKey(secret)

.parseClaimsJws(token)

.getBody();

User u = new User();

u.setUsername(body.getSubject());

u.setId(Long.parseLong((String) body.get("userId")));

u.setRole((String) body.get("role"));

return u;

} catch (JwtException | ClassCastException e) {

return null;

}

}

/\*\*

\* Generates a JWT token containing username as subject, and userId and role as additional claims. These properties are taken from the specified

\* User object. Tokens validity is infinite.

\*

\* @param u the user for which the token will be generated

\* @return the JWT token

\*/

public String generateToken(User u) {

Claims claims = Jwts.claims().setSubject(u.getUsername());

claims.put("userId", u.getId() + "");

claims.put("role", u.getRole());

return Jwts.builder()

.setClaims(claims)

.signWith(SignatureAlgorithm.HS512, secret)

.compact();

}

}

//the encoded JWT

eyJhbGciOiJSUzI1NiJ9

.eyJzdWIiOiJ1c2VyMSJ9

.HG9GCQPuC6w6pulbYE2uurCzpEwoWvz\_8Ps5ZjgtfomyY4LWacDEzlHLnyMj9H7aqgcePC7\_4l2wDXQV-S0BQRsIZfJeUUmWxlTlLzvKZr\_2eEx00YZPPFZNoFCfwB-ajLHLLenROy4aSjPo\_Vg9o7N-p0DZ1yZQoJhkvoVJgkhX9FeAf65kIZkbuJC9dmVkzXSOpVf4GZeCpNDJJYSo6IAnL3UEoWek6V9BtWgV-a4xvydp7vxkdDXmzmalGLYuWbuVG7rWcbWwSfsg38iEG-mqptqA\_Kzk1VmjwWNo\_BfvLuzjzuosqi732-5SRzBP-2zqGghBqMYsGgkqkH2n7A

//human readable format

{

"alg": "RS256" //header

}

{

"sub": "user1" //claim payload

}

Finally, the JwtUtil class is in charge of parsing the token into User object and generating the token from the User object. It is straightforward since it uses the [jjwt library](https://github.com/jwtk/jjwt) to do all the JWT work. In our example, we simply store the username, user ID and user roles in the token. We could also store more arbitrary stuff and add more security features, such as the token’s expiration. Parsing of the token is used in the AuthenticationProvider as shown above. The generateToken() method is called from login and signup REST services, which are unsecured and will not trigger any security checks or require a token to be present in the request. In the end, it generates the token that will be returned to the clients, based on the user.

**Conclusion**

Although the old, standardized security approaches (session cookie, HTTP Basic, and HTTP Digest) will work with REST services as well, they all have problems that would be nice to avoid by using a better standard. JWT arrives just in time to save the day, and most importantly it is very close to becoming an IETF standard.

JWT’s main strength is handling user authentication in a stateless, and therefore scalable, way, while keeping everything secure with up-to-date cryptography standards. Storing claims (user roles and permissions) in the token itself creates huge benefits in distributed system architectures where the server that issues the request has no access to the authentication data source.