**How do you explain distributed applications?**

For me, it’s an amazing thing to see an application work across dozens (even hundreds) of servers. It’s like watching an orchestra playing a piece of music. While the final product of an orchestra is beautiful, the making of it is often a lot of hard work and requires a significant amount of practice. The same goes for writing a massively distributed application.

Since I entered the software development field 25 years ago, I’ve watched the industry struggle with the “right” way to build distributed applications. I’ve seen distributed service standards such as CORBA rise and fall. Monstrously big companies have tried to push big and, often, proprietary protocols. Anyone remember Microsoft’s Distributed Component Object Model (DCOM) or Oracle’s J2EE’s Enterprise Java Beans 2 (EJB)? I watched as technology companies and their followers rushed to build service-oriented architectures (SOA) using heavy XML-based schemas.

**What is a microservice?**

 A microservice architecture focuses on building small services that use simple protocols (HTTP and JSON) to communicate. That’s it. You can write a microservice with nearly any programming language. There’s beauty in this simplicity.

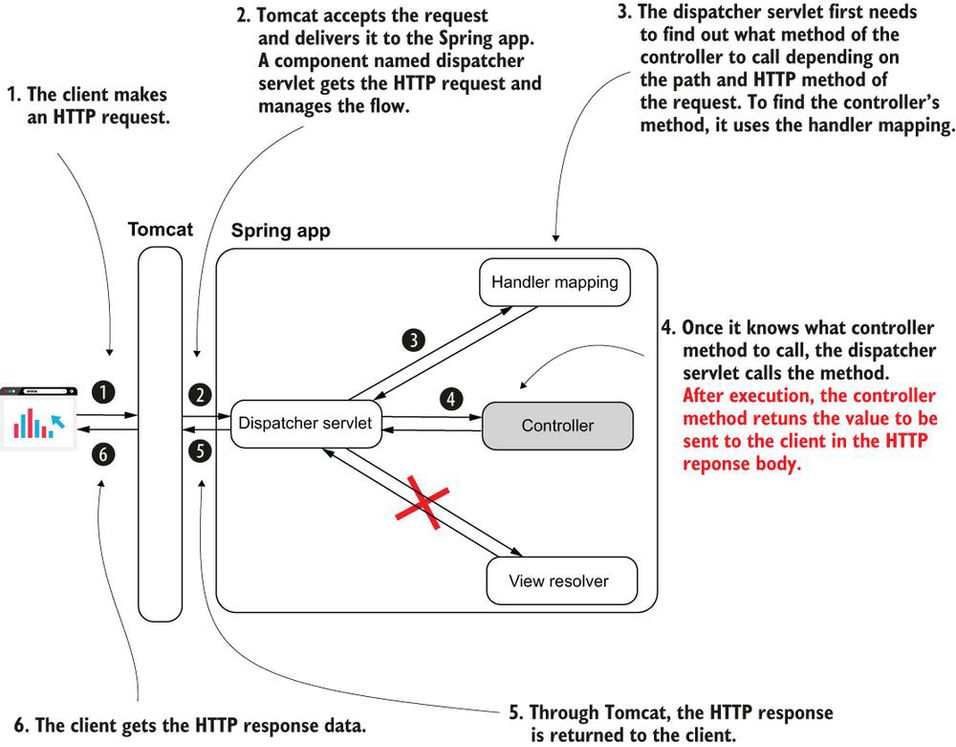
**Why Spring cloud came into being?**

The Spring Cloud framework provides out-of-the-box solutions for many of the common development and operational problems you’ll run into as a microservice developer. Spring Cloud lets you use only the pieces you need and minimizes the amount of work you need to do to build and deploy production-ready Java micro-services. It does this by using other battle-hardened technologies from companies and groups such as Netflix, HashiCorp, and the Apache foundation.

Building an individual microservice is easy, operationalizing and scaling it is difficult. Getting hundreds of small distributed components to work together and then building a resilient application from them can be incredibly difficult to do. In distributed computing, failure is a fact of life and how your application deals with it is incredibly difficult to get right. To paraphrase my colleagues Chris Miller and Shawn Hagwood: “If it’s not breaking once in a while, you’re not building.”

**What are REST web services?**

When implementing REST endpoints, the Spring MVC flow changes. The app no longer needs a view resolver because the client needs the data returned by the controller's action directly. Once the controller's action completes, the dispatcher servlet returns the HTTP response without rendering any view.



**What are some challenges REST endpoint bring?**

* If the controller's action takes a long time to complete, the HTTP call to the endpoint might time out and break the communication.
* Sending a large quantity of data in one call (through the HTTP request) might cause the call to time out and break the communication. Sending more than a few megabytes through a REST call usually isn't the right choice.
* Too many concurrent calls on an endpoint exposed by a backend component might put too much pressure on the app and cause it to fail.
* The network supports the HTTP calls, and the network is never 100% reliable. There's always a chance a REST endpoint call might fail because of the network.

**Does REST endpoint implemented using Spring internally uses Spring MVC mechanism?**

Yes

**What does @ResponseBody annotation do?**

The @ResponseBody annotation tells the dispatcher servlet that the controller's action doesn't return a view name but the data sent directly in the HTTP response.

**What will happen if do not provide @ResponseBody annotation in your REST controller method?**

Spring will try to resolve view based returned data and will give runtime exception related view resolving.

**Why do we need @RestController annotation?**

So that you don’t have repeat @ResponseBody annotation on every method?

@RestController basically combination of @Controller and @ResponseBody.

Without @RestController:

@Controller

public class HelloController {

@GetMapping("/hello")

@ResponseBody

public String hello() {

return "Hello!";

}

@GetMapping("/ciao")

@ResponseBody

public String ciao() {

return "Ciao!";

}

}

With @RestController:

@RestController ❶

public class HelloController {

@GetMapping("/hello")

public String hello() {

return "Hello!";

}

@GetMapping("/ciao")

public String ciao() {

return "Ciao!";

}

}

❶ Instead of repeating the @ResponseBody annotation for each method, we replace @Controller with @RestController.

**What are different tools available to test your REST endpoints?**

* Postman—Offers a nice GUI and is comfortable to use
* cURL—A command-line tool useful in cases where you don't have a GUI (e.g., when you connect to a virtual machine via SSH or when you write a batch script)

Both these tools are a must-learn for any developer.

**What is HTTP response made of?**

The HTTP response holds data as the following:

* Response headers—Short pieces of data in the response (usually not more than a few words long)
* The response body—A larger amount of data the backend needs to send in the response.
* The response status—A short representation of the request's result

**Can you give typical example of how you implement REST endpoint?**

public class Country {

private String name;

private int population;

public static Country of( ❶

String name,

int population) {

Country country = new Country();

country.setName(name);

country.setPopulation(population);

return country;

}

// Omitted getters and setters

}

@RestController ❶

public class CountryController {

@GetMapping("/france") ❷

public Country france() {

Country c = Country.of("France", 67);

return c; ❸

}

}

❶ Marking the class as a REST controller to add a bean in the Spring context and also inform the dispatcher servlet not to look for a view when this method returns

❷ Mapping the controller's action to the HTTP GET method and /france path

❸ Returning an instance of type Country

**What is default status code in Spring for REST endpoints?**

By default, Spring sets some common HTTP statuses:

* 200 OK if no exception was thrown on the server side while processing the request.
* 404 Not Found if the requested resource doesn't exist.
* 400 Bad Request if a part of the request could not be matched with the way the server expected the data.
* 500 Error on server if an exception was thrown on the server side for any reason while processing the request. Usually, for this kind of exception, the client can't do anything, and it's expected someone should solve the problem on the backend.

**How do you configure custom status code?**

The easiest and most common way to customize the HTTP response is using the ResponseEntity class. This class provided by Spring allows you to specify the response body, status, and headers on the HTTP response.

@RestController

public class CountryController {

@GetMapping("/france")

public ResponseEntity<Country> france() {

Country c = Country.of("France", 67);

return ResponseEntity

.status(HttpStatus.ACCEPTED) ❶

.header("continent", "Europe") ❷

.header("capital", "Paris") ❷

.header("favorite\_food", "cheese and wine") ❷

.body(c); ❸

}

}

❶ Changes the HTTP response status to 202 Accepted

❷ Adds three custom headers to the response

❸ Sets the response body

**What are two ways of handling exceptions for Spring REST APIs?**

**1.** Managing the HTTP response for exceptions in the controller's action

@RestController

public class PaymentController {

private final PaymentService paymentService;

public PaymentController(PaymentService paymentService) {

this.paymentService = paymentService;

}

@PostMapping("/payment")

public ResponseEntity<?> makePayment() {

try {

PaymentDetails paymentDetails = ❶

paymentService.processPayment();

return ResponseEntity ❷

.status(HttpStatus.ACCEPTED)

.body(paymentDetails);

} catch (NotEnoughMoneyException e) {

ErrorDetails errorDetails = new ErrorDetails();

errorDetails.setMessage("Not enough money to make the payment.");

return ResponseEntity ❸

.badRequest()

.body(errorDetails);

}

}

}

❶ We try calling the processPayment() method of the service.

❷ If calling the service method succeeds, we return an HTTP response with status Accepted and the PaymentDetails instance as a response body.

❸ If an exception of type NotEnoughMoneyException is thrown, we return an HTTP response with status Bad Request and an ErrorDetails instance as a body.

2. Separating the exception logic with a REST controller advice

@RestControllerAdvice ❶

public class ExceptionControllerAdvice {

@ExceptionHandler(NotEnoughMoneyException.class) ❷

public ResponseEntity<ErrorDetails> exceptionNotEnoughMoneyHandler() {

ErrorDetails errorDetails = new ErrorDetails();

errorDetails.setMessage("Not enough money to make the payment.");

return ResponseEntity

.badRequest()

.body(errorDetails);

}

}

❶ We use the @RestControllerAdvice annotation to mark the class as a REST controller advice.

❷ We use the @ExceptionHandler method to associate an exception with the logic the method implements.

**Can an HTTP GET endpoint use a request body?**

Before 2014, the HTTP protocol specification didn't allow a request body for HTTP GET calls. No implementation for the client or server side allowed you to use a request body with an HTTP GET call.

The HTTP specification changed in 2014, and it now allows the use of the request body with an HTTP GET call. But sometimes students find old articles on the internet or read book editions that were not updated, and this seems to create confusion years later.

**What are three different ways to call an API endpoint?**

1. **OpenFeign**—A tool offered by the Spring Cloud project. I recommend developers use this feature in new apps for consuming REST endpoints.
2. **RestTemplate**—A well-known tool developers have used since Spring 3 to call REST endpoints. RestTemplate is often used today in Spring apps. However, as we'll discuss in this chapter, OpenFeign is a better alternative to RestTemplate, so if you work on a new app, you'll probably avoid RestTemplate and use OpenFeign instead.
3. **WebClient**—A Spring feature presented as an alternative to RestTemplate. This feature uses a different programming approach named reactive programming, which we'll discuss at the end of this chapter.

**Will Spring’s RestTemplate be deprecated eventually?**

Yes.

RestTemplate has been put in maintenance mode starting with Spring 5, and it will eventually be deprecated.

**When you are using OpenFeign, how much is learning curve for OpenFeign?**

OpenFeign uses the same annotations that you normally use to create contronller methods?

You don't have to learn something specific to OpenFeign. Just use the same annotations as for exposing the REST endpoints in the Spring MVC controller classes.

**Why OpenFeign is considered a better option that RestClient?**

I don't want you to conclude that RestTemplate has any problems. It is being put to sleep not because it's not working properly or because it's not a good tool. But as apps evolved, we started to need more capabilities. Developers wanted to be able to benefit from different things that aren't easy to implement with RestTemplate, such as the following:

* Calling the endpoints both synchronously and asynchronously
* Writing less code and treating fewer exceptions (eliminate boilerplate code)
* Retrying call executions and implementing fallback operations (logic performed when the app can't execute a specific REST call for any reason)

In other words, developers prefer to get more things out of the box rather than implement them wherever possible.

**What is core purpose of frameworks?**

Reusing code and avoiding boilerplate code is one of the primary purposes of frameworks.

**What is one reason to learn something which is deprecated or considered legacy code?**

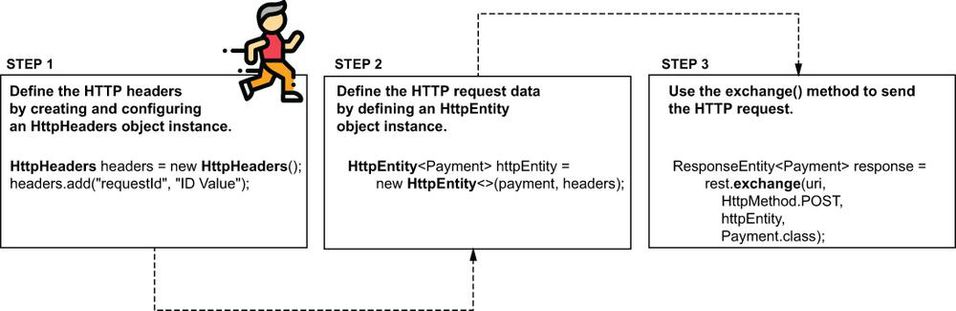
When something is called “deprecated” or “legacy,” it doesn't necessarily mean you shouldn't learn it. Sometimes, deprecated technologies are still used in projects many years after being declared deprecated, including RestTemplate and the Spring Security OAuth project.

**What are main steps when calling a backend API using RestTemplate?**

The steps for defining the call are as follows ([figure 11.6](https://cdn2.percipio.com/1693821852.1e95f63bbef1e31025db72cfe5d70f8ad38d8658/eod/books/158139/OEBPS/section-112-64.xhtml#ch11fig06)):

1. Define the HTTP headers by creating and configuring an HttpHeaders instance.
2. Create an HttpEntity instance that represents the request data (headers and body).
3. Send the HTTP call using the exchange() method and get the HTTP response.

Figure 11.6: To define a more complex HTTP request, you have to use the HttpHeaders class to define the headers, then the HttpEntity class to represent the full request data. Once you defined the data on the request, you call the exchange() method to send it

Larger View

**What is high level difference between URL and URI?**

[**http://localhost:8080/**](http://localhost:8080/) **is URL**

[**http://localhost:8080/payment**](http://localhost:8080/payment) **id URL**

**For example,**

@Component

public class PaymentsProxy {

private final RestTemplate rest;

@Value("${name.service.url}")

private String paymentsServiceUrl; ❶

public PaymentsProxy(RestTemplate rest) { ❷

this.rest = rest;

}

public Payment createPayment(Payment payment) {

String uri = paymentsServiceUrl + "/payment";

HttpHeaders headers = new HttpHeaders(); ❸

headers.add("requestId", ❸

UUID.randomUUID().toString()); ❸

HttpEntity<Payment> httpEntity = ❹

new HttpEntity<>(payment, headers);

ResponseEntity<Payment> response = ❺

rest.exchange(uri, ❺

HttpMethod.POST, ❺

httpEntity, ❺

Payment.class); ❺

return response.getBody(); ❻

}

}

❶ We take the URL to the payment service from the properties file.

❷ We inject the RestTemplate from the Spring context using constructor DI.

❸ We build the HttpHeaders object to define the HTTP request headers.

❹ We build the HttpEntity object to define the request data.

❺ We send the HTTP request and retrieve the data on the HTTP response.

❻ We return the HTTP response body.

**What is HttpEntity?**

It is combination of HttpHeaders and Body.

public Payment createPayment(Payment payment) {

String uri = paymentsServiceUrl + "/payment";

HttpHeaders headers = new HttpHeaders(); ❸

headers.add("requestId", ❸

UUID.randomUUID().toString()); ❸

HttpEntity<Payment> httpEntity = ❹

new HttpEntity<>(payment, headers);

ResponseEntity<Payment> response = ❺

rest.exchange(uri, ❺

HttpMethod.POST, ❺

httpEntity, ❺

Payment.class); ❺

return response.getBody(); ❻

}

**What is non-reactive app or app which not reactive app?**

In a nonreactive app, a thread executes a business flow. Multiple tasks compose a business flow, but these tasks are not independent. The same thread executes all the tasks composing a flow.

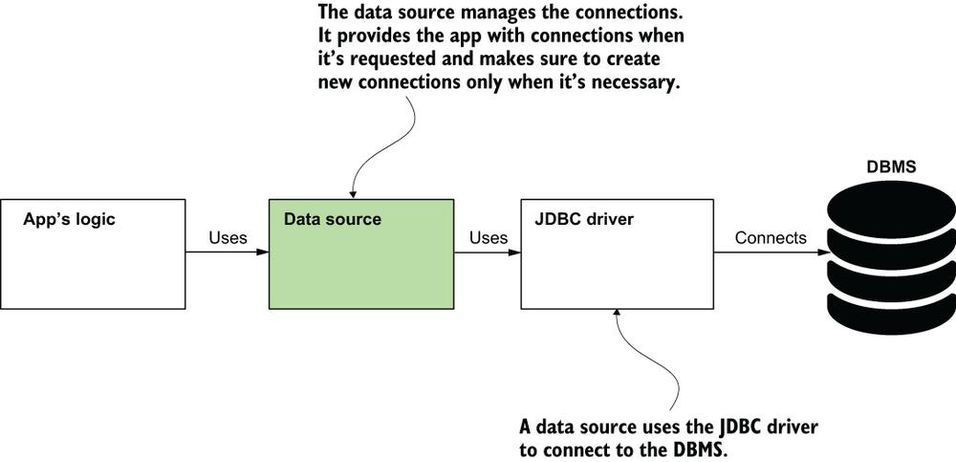
**How do you make your Spring application work with databases?**

To make your Spring app work with a database. The straightforward way is the JdbcTemplate tool that Spring offers.

**What is data source?**

The data source ([figure 12.2](https://cdn2.percipio.com/1693995341.4c7b5ce26eca51d5dfb0f2b5a07017bb7f754a2f/eod/books/158139/OEBPS/section-121-68.xhtml#ch12fig02)) is a component that manages connections to the server handling the database(the database management system, also known as DBMS).

The data source is a component that manages connections to the database management systems (DBMS). The data source uses the JDBC driver to get the connections it manages. The data source aims to improve the app's performance by allowing its logic to reuse connections to the DBMS and request new connections only when it needs them. The data source also makes sure to close the connections when it releases them

Larger View

Without an object taking the responsibility of a data source, the app would need to request a new connection for each operation with the data. This approach is not realistic in a production scenario because communicating through the network for establishing a new connection for each operation would dramatically slow down the application and cause performance issues. The data source makes sure your app only requests a new connection when it really needs it, improving the app's performance.

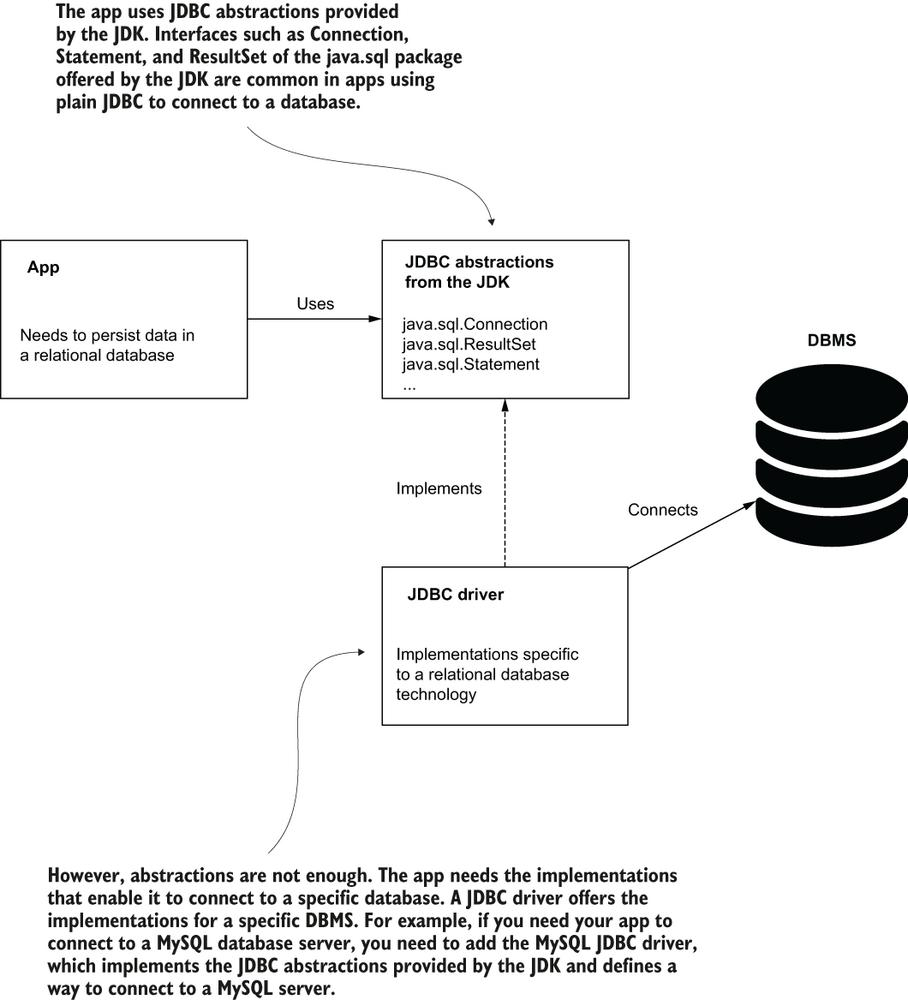
**What is JDBC (Java Database Connectivity)?**

The language's capabilities to connect to a relational database is named Java Database Connectivity (JDBC). JDBC offers you a way to connect to a DBMS to work with a database.

**What is Database Driver?**

The JDK doesn't provide a specific implementation for working with a particular technology (such as MySQL, Postgres, or Oracle). The JDK only gives you the abstractions for objects an app needs to work with a relational database. To gain the implementation of this abstraction and enable your app to connect to a certain DBMS technology, you add a runtime dependency named the JDBC driver ([figure 12.3](https://cdn2.percipio.com/1693995341.4c7b5ce26eca51d5dfb0f2b5a07017bb7f754a2f/eod/books/158139/OEBPS/section-121-68.xhtml#ch12fig03)). Every technology vendor provides the JDBC driver you need to add to your app to enable it to connect to that specific technology. The JDBC driver is not something that comes either from the JDK or from a framework such as Spring.

When connecting to a database, a Java app uses JDBC. The JDK provides a set of abstractions, but the app needs a certain implementation that depends on the relational database technology the app connects to. A runtime dependency named JDBC driver offers these implementations. For each specific technology, such a driver exists, and the app needs the exact driver that offers the implementations for the server technology it needs to connect to.



**What are different ways applications connecting to database systems?**

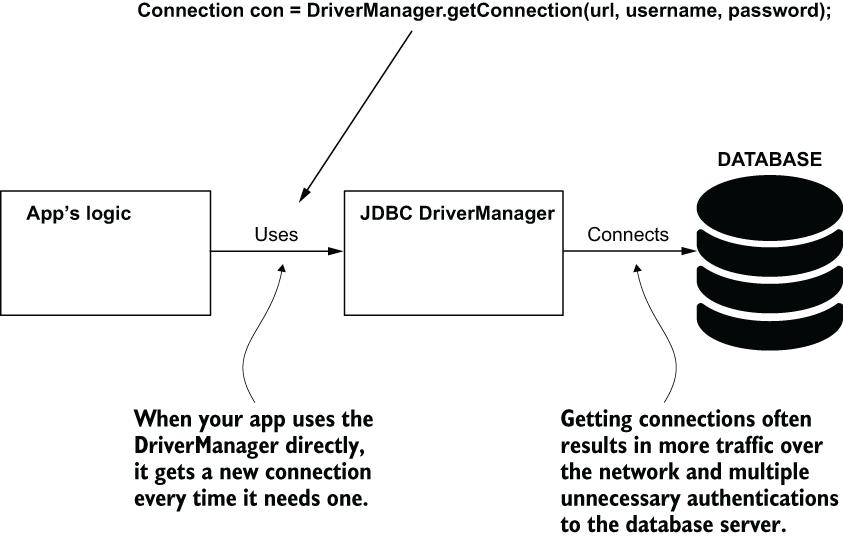
1. Using directly DriverManager

Connection con = DriverManager.getConnection(url, username, password);

A first option is to use the JDBC driver directly and implement your app to require a connection each time it needs to execute a new operation on the persisted data. You'll often find this approach in Java fundamentals tutorials.

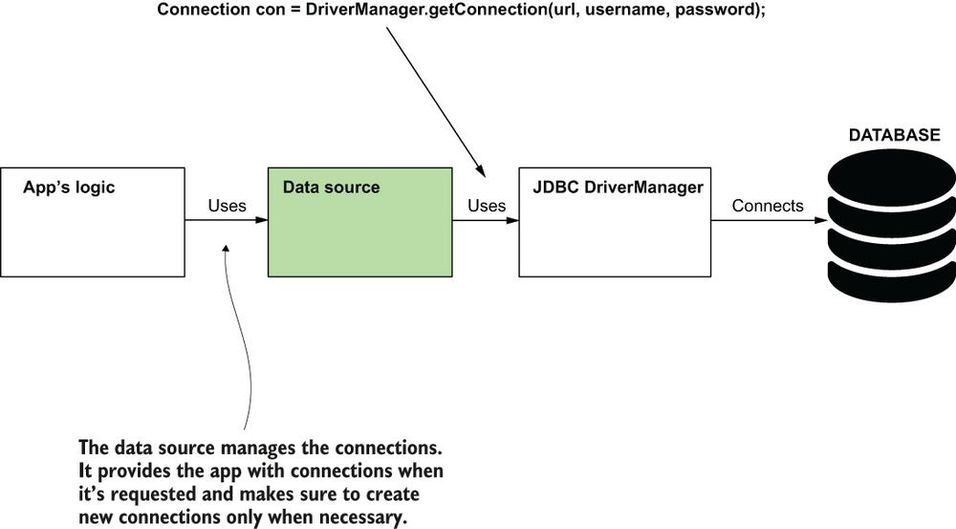
The getConnection() method uses the URL provided as a value for the first parameter to identify the database your app needs to access and the username and password to authenticate the access to the database.

But requesting a new connection and authenticating each operation again and again for each is a waste of resources and time for both the client and the database server. Imagine you go into a bar and ask for a beer; you look young, so the barman asks for your ID. This is fine, but it would become tedious if the barman asked you for the ID again when you ordered the second and the third beer (hypothetically, of course).



2. Using data source

Adding a data source to the class design helps the app spare the time for unnecessary operations. The data source manages the connections, provides the app with connections when requested, and creates new connections only when needed.



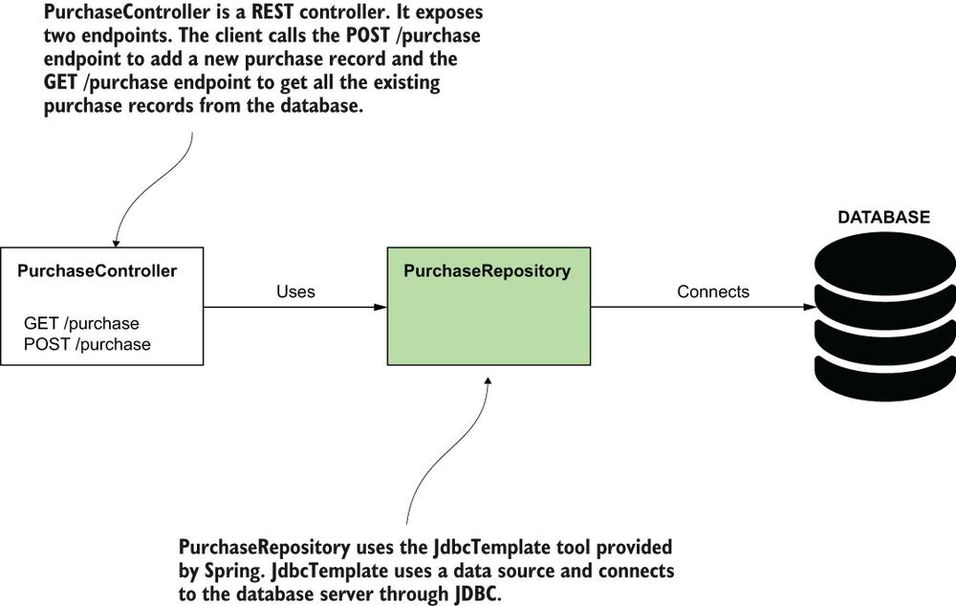
**What is JdbcTemplate in Spring?**

JdbcTemplate is the simplest of the tools Spring offers for using a relational database, but it's an excellent choice for small apps as it doesn't force you to use any other specific persistence framework. JdbcTemplate is the best Spring choice to implement a persistence layer when you don't want your app to have any other dependency. I also consider it an excellent way to start learning how to implement the persistence layer of Spring apps.

**What is repository. Explain with a typical example?**

A repository is a class responsible with working with a database.

A REST controller implements two endpoints. When a client calls the endpoints, the controller delegates to a repository object to use the database.



We start the implementation as usual, by adding the necessary dependencies. The next code snippet shows you the dependencies you need to add as they appear in the project's pom.xml file:

<dependency> ❶

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

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

</dependency> ❶

<dependency> ❷

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

<artifactId>spring-boot-starter-jdbc</artifactId> ❷

</dependency> ❷

<dependency> ❸

<groupId>com.h2database</groupId> ❸

<artifactId>h2</artifactId> ❸

<scope>runtime</scope> ❹

</dependency>

❶ We use the same web dependency as we did in previous chapters to implement the REST endpoints.

❷ We add the JDBC starter to get all the needed capabilities to work with databases using JDBC.

❸ We add the H2 dependency to get both an in-memory database for this example and a JDBC driver to work with it.

❹ The app only needs the database and the JDBC driver at runtime. The app doesn't need them for compilation. To instruct Maven we only want these dependencies at runtime, we add the scope tag with the value “runtime.”

**What are different options available for maintaining database schema versioning?**

1. **Flyway**
2. **Liquibase**

Using a “schema.sql” file to define the database structure only works for theoretical examples. This approach is easy because it's fast and allows you to focus on the things you learn rather than the definition of the database structure in a tutorial. But in a real-world example, you will need to use a dependency that also allows you to version your database scripts. I recommend you look at Flyway (<https://flywaydb.org/>) and Liquibase (<https://www.liquibase.org/>). These are two highly appreciated dependencies for database schema versioning.

**What datatype should for prices?**

When operating with double and float values, you might lose precision for even simple arithmetic operations such as addition or subtraction. This effect is caused by the way Java stores such values in memory. When you work with sensitive information such as prices, you should use the BigDecimal type instead. Don't worry about the conversion. All the essential capabilities Spring provides know how to use BigDecimal.

When you want to store a floating-point value accurately and make sure you don't lose decimal precision when executing various operations with the values, use BigDecimal and not double or float.

**How autoconfiguration in Spring boot works when working databases?**

Like in many production scenarios, we'll benefit once more from Spring Boot's magic. When Spring Boot saw you added the H2 dependency in pom.xml, it automatically configured a data source and a JdbcTemplate instance. In this example, we'll use them directly.

If you use Spring but not Spring Boot, you need to define the DataSource bean and the JdbcTemplate bean (you can add them in the Spring context using the @Bean annotation in the configuration class, as you learned in [chapter 2](https://cdn2.percipio.com/1694001144.14619cde8b46c6171b7e964ccfdd467f4c67923d/eod/books/158139/OEBPS/chapter-2-17.xhtml#ch02)). In [section 12.3](https://cdn2.percipio.com/1694001144.14619cde8b46c6171b7e964ccfdd467f4c67923d/eod/books/158139/OEBPS/section-123-70.xhtml#ch12lev1sec4), I'll show you how to customize them and for which scenarios you need to define your own data source and JdbcTemplate instances.

@Repository

public class PurchaseRepository {

private final JdbcTemplate jdbc;

public PurchaseRepository(JdbcTemplate jdbc) {

this.jdbc = jdbc;

}

public void storePurchase(Purchase purchase) { ❶

String sql = ❷

"INSERT INTO purchase VALUES (NULL, ?, ?)";

jdbc.update(sql, ❸

purchase.getProduct(),

purchase.getPrice());

}

}

❶ We use constructor injection to get the JdbcTemplate instance from the application context.

❶ The method takes a parameter that represents the data to be stored.

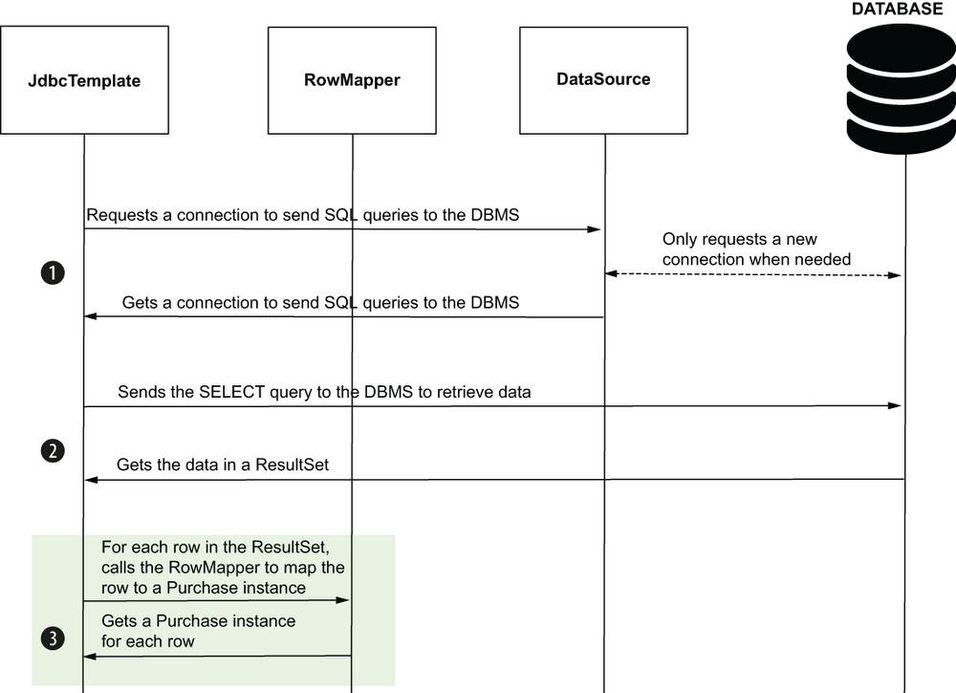
❷ The query is written as a string, and question marks (?) replace the queries’ parameter values. For the ID, we use NULL because we configured the DBMS to generate the value for this column.

❸ The JdbcTemplate update() method sends the query to the database server. The first parameter the method gets is the query, and the next parameters are the values for the parameters. These values replace, in the same order, each question mark in the query.

dbcTemplate has an update() method you can use to execute any query for data mutation: INSERT, UPDATE or DELETE. Pass the SQL and the parameters it needs, and that's it; let JdbcTemplate take care of the rest (obtaining a connection, creating a statement, treating the SQLException, and so on). The following listing adds a storePurchase() method to the PurchaseRepository class. The storePurchase() method uses JdbcTemplate to add a new record in the purchase table.

**How do you map ResultSet to Java Object?**

JdbcTemplate uses the RowMapper to change the ResultSet to a list of Purchase instances. For each row in the ResultSet, JdbcTemplate calls the RowMapper to map the row to a Purchase instance. The diagram presents all three steps JdbcTemplate follows to send the SELECT query: (1) get a DBMS connection, (2) send the query and retrieve the result, and (3) map the result to Purchase instances.



@Repository

public class PurchaseRepository {

// Omitted code

public List<Purchase> findAllPurchases() { ❶

String sql = "SELECT \* FROM purchase"; ❷

RowMapper<Purchase> purchaseRowMapper = (r, i) -> { ❸

Purchase rowObject = new Purchase(); ❹

rowObject.setId(r.getInt("id")); ❹

rowObject.setProduct(r.getString("product")); ❹

rowObject.setPrice(r.getBigDecimal("price")); ❹

return rowObject; ❹

};

return jdbc.query(sql, purchaseRowMapper); ❺

}

}

❶ The method returns the records it retrieves from the database in a list of Purchase objects.

❷ We define the SELECT query to get all the records from the purchase table.

❸ We implement a RowMapper object that tells JdbcTemplate how to map a row in the result set into a Purchase object. In the lambda expression, parameter “r” is the ResultSet (the data you get from the database), while parameter “i” is an int representing the row number.

❹ We set the data into a Purchase instance. JdbcTemplate will use this logic for each row in the result set.

❺ We send the SELECT query using the query method, and we provide the row mapper object for JdbcTemplate to know how to transform the data it gets in Purchase objects.

**What is best practice regarding storing database password for Spring boot apps?**

In Secret Vaults.

**What are the scenarios in which we need to define data source bean by ourselves?**

Spring Boot knows how to use a DataSource bean if you provide the connection details in the “application.properties” file. Sometimes this is enough, and as usual, I recommend you go with the simplest solution that solves your problems. But in other cases, you can't rely on Spring Boot to create your DataSource bean. In such a case, you need to define the bean yourself. Some scenarios in which you need to define the bean yourself are as follows:

* You need to use a specific DataSource implementation based on a condition you can only get at runtime.
* Your app connects to more than one database, so you have to create multiple data sources and distinguish them using qualifiers.
* You have to configure specific parameters of the DataSource object in certain conditions your app has only at runtime. For example, depending on the environment where you start the app, you want to have more or fewer connections in the connection pool for performance optimizations.
* Your app uses Spring framework but not Spring Boot.
* @Configuration
* public class ProjectConfig {
* @Value("${custom.datasource.url}") ❶
* private String datasourceUrl;
* @Value("${custom.datasource.username}") ❶
* private String datasourceUsername;
* @Value("${custom.datasource.password}") ❶
* private String datasourcePassword;
* @Bean ❷
* public DataSource dataSource() { ❸
* HikariDataSource dataSource = ❹
* new HikariDataSource();
* dataSource.setJdbcUrl(datasourceUrl); ❺
* dataSource.setUsername(datasourceUsername); ❺
* dataSource.setPassword(datasourcePassword); ❺
* dataSource.setConnectionTimeout(1000); ❻
* return dataSource; ❼
* }
* }
* ❶ The connection details are configurable, so it's a good idea to continue defining them outside of the source code. In this example, we keep them in the “application.properties” file.
* ❷ We annotate the method with @Bean to instruct Spring to add the returned value to its context.
* ❸ The method returns a DataSource object. If Spring Boot finds a DataSource already exists in the Spring context it doesn't configure one.
* ❹ We'll use HikariCP as the data source implementation for this example. However, when you define the bean yourself, you can choose other implementations if your project requires something else.
* ❺ We set the connection parameters on the data source.
* ❻ You can configure other properties as well (eventually in certain conditions). In this case, I use the connection timeout (how much time the data source waits for a connection before considering it can't get one) as an example.
* ❼ We return the DataSource instance, and Spring adds it to its context.

Don't forget to configure values for the properties you inject using the @Value annotation. In the “application.properties” file these properties should look like the next code snippet. I have intentionally used the word “custom” in their name to stress that we chose these names, and they're not Spring Boot properties. You can give these properties any name:

custom.datasource.url=jdbc:mysql://localhost/spring\_quickly?

useLegacyDatetimeCode=false&serverTimezone=UTC

custom.datasource.username=root

custom.datasource.password=

**What is Transaction?**

A transaction is a defined set of mutable operations (operations that change data) that can either correctly execute them altogether or not at all. We refer to this as atomicity. Transactions are essential in apps because they ensure the data remains consistent if any step of the use case fails when the app already changed data.

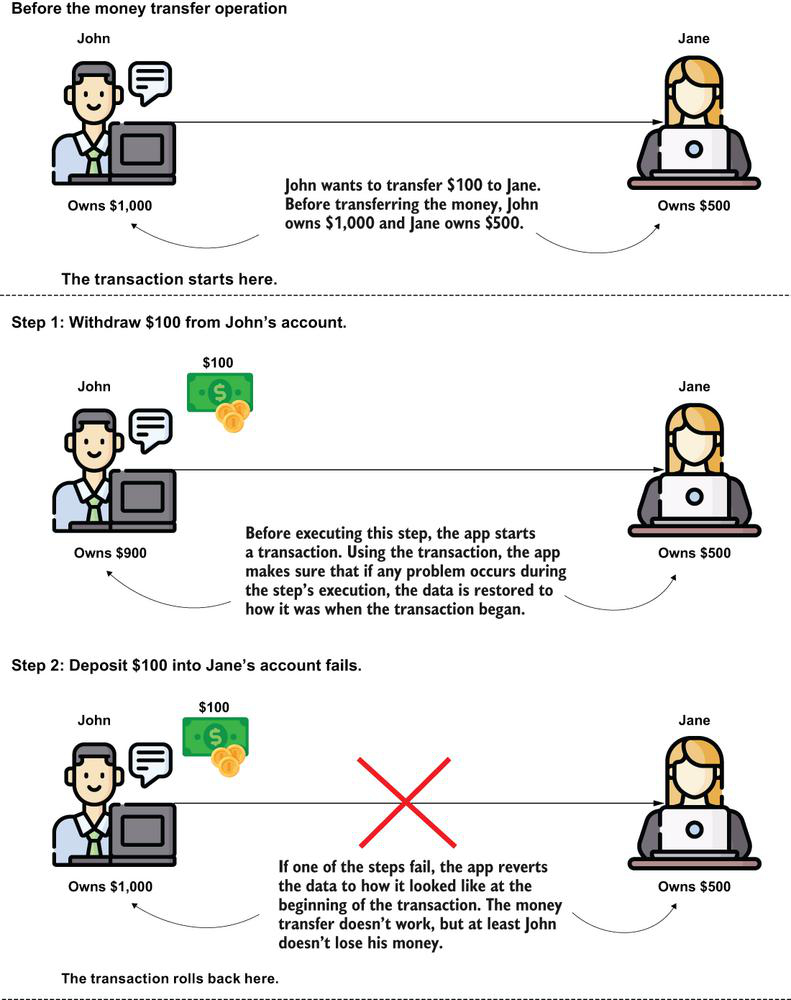
Let's again consider a (simplified) transfer money functionality consisting of two steps:

1. Withdraw money from the source account.
2. Deposit money into the destination account.

We can start a transaction before step 1 and close the transaction after step 2.

In such a case, if both steps successfully execute, when the transaction ends (after step 2), the app persists the changes made by both steps. We also say, in this case, that the transaction “commits.” The “commit” operation happens when the transaction ends and all the steps are successfully executed, so the app persists the data changes.

A transaction solves possible inconsistencies that could appear if any of the steps of a use case fail. With a transaction, if any of the steps fail, the data is reverted to how it was at the transaction start.



**What is Commit?**

The successful end of a transaction when the app stores all the changes made by the transaction's mutable operations.

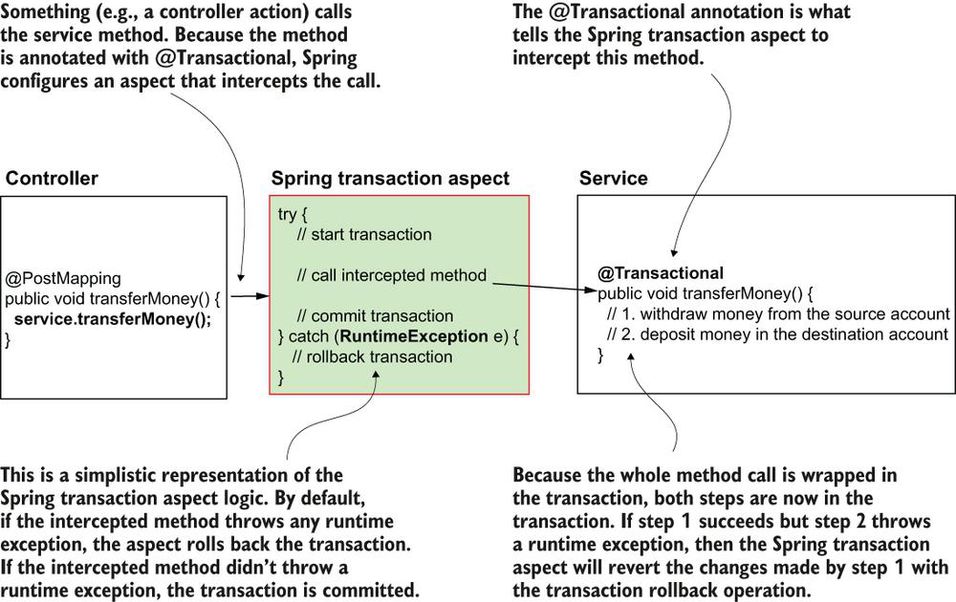
**What is Rollback?**

The transaction ends with rollback when the app restores the data to the way it looked at the beginning of the transaction to avoid data inconsistencies.

**How are aspects related to Transactions?**

An aspect is a piece of code that intercepts specific methods’ execution in a way that you define. In most cases today, we use annotations to mark the methods whose execution an aspect should intercept and alter. For Spring transactions, things aren't different. To mark a method we want Spring to wrap in a transaction, we use an annotation named @Transactional. Behind the scenes, Spring configures an aspect (you don't implement this aspect yourself; Spring provides it) and applies the transaction logic for the operations executed by that method.

When you use the @Transactional annotation with a method, an aspect configured by Spring intercepts the method call and applies the transaction logic for that call. The app doesn't persist the changes the method makes if the method throws a runtime exception



Spring knows to rollback a transaction if the method throws a runtime exception. But I'd like to emphasize the word “throws.” When I teach Spring in class, students often understand that it's enough that some operation inside the transferMoney() method throws a runtime exception. But this is not enough! The transactional method should throw the exception further so that the aspect knows it should rollback the changes. If the method treats the exception in its logic and doesn't throw the exception further, the aspect can't know the exception occurred.

**How does Spring handles transactions for checked exceptions?**

Checked exceptions in Java are those exceptions you have to treat or throw; otherwise, your app won't compile. Do they also cause a transaction rollback if a method throws them? By default, no! Spring's default behavior is only to roll back a transaction when it encounters a runtime exception. This is how you'll find transactions used in almost all real-world scenarios.

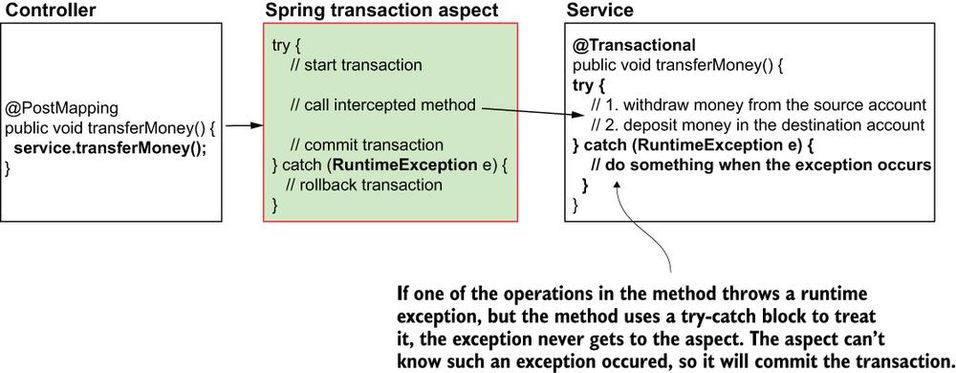
When you work with a checked exception, you have to add the “throws” clause in the method signature; otherwise, your code won't compile, so you always know when your logic could throw such an exception. For this reason, a situation represented with a checked exception is not an issue that could cause data inconsistency, but is instead a controlled scenario that should be managed by the logic the developer implements.

If, however, you'd like Spring to also roll back transactions for checked exceptions, you can alter Spring's default behavior. The @Transactional annotation, which you'll learn to use in [section 13.3](https://cdn2.percipio.com/1694021919.1f07dff8314c46d65877481f2156fef3ddf25640/eod/books/158139/OEBPS/section-133-75.xhtml#ch13lev1sec4), has attributes for defining which exceptions you want Spring to roll back the transactions for.

However, I recommend you always keep your application simple and, unless needed, rely on the framework's default behaviour.

**What will happen if you handle runtime exception in the method. Will @transactional work in that scenario?**

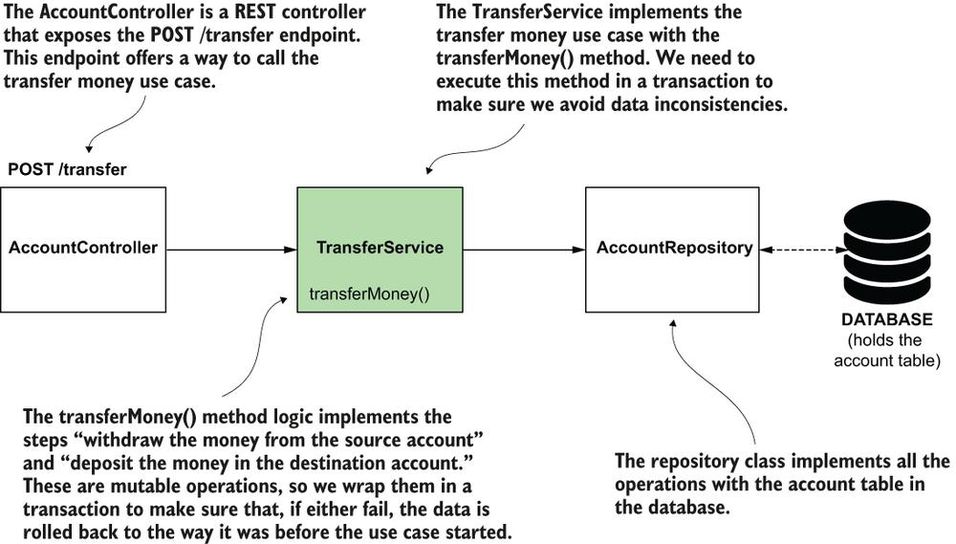
If a runtime exception is thrown inside the method, but the method treats the exception and doesn't throw it back to the caller, the aspect won't get this exception and will commit the transaction. When you treat an exception in a transactional method, such as in this case, you need to be aware the transaction won't be rolled back, as the aspect managing the transaction cannot see the exception.



**How do you handle transactions in a Spring application?**

Declaring a transaction in a Spring app is as easy as using an annotation: @Transactional. You use @Transactional to mark a method you want Spring to wrap in a transaction. You don't need to do anything else. Spring configures an aspect that intercepts the methods you annotate with @Transactional. This aspect starts a transaction and either commits the method's changes if everything went fine or rolls back the changes if any runtime exception occurred.

We implement the transfer money use case in a service class and expose this service method through a REST endpoint. The service method uses a repository to access the data in the database and change it. The service method (which implements the business logic) must be wrapped in a transaction to avoid data inconsistencies if problems occur during the method execution.



**Can we use @Transactional annotation at class level as well? What will happen if @Transactional annotation is applied on both class level and method level?**

The @Transactional annotation can also be applied directly to the class. If used on the class (as presented in the next code snippet), the annotation applies to all the class methods. Often in real-world apps you will find the @Transactional annotation used on the class, because the methods of a service class define use cases and, in general, all the use cases need to be transactional. To avoid repeating the annotation on each method, it's easier just to mark the class once. When using @Transactional on both the class and the method, the method level's configuration overrides the one on the class.

@Service

@Transactional ❶

public class TransferService {

// Omitted code

public void transferMoney(long idSender,

long idReceiver,

BigDecimal amount) {

// Omitted code

}

}

❶ We often use the @Transactional annotation directly with the class. If the class has multiple methods, @Transactional applies to all of them.

**What are DTOs?**

The TransferRequest object simply models the HTTP request body. Such objects, whose responsibility is to model the data transferred between two apps, are DTOs. The following listing shows the definition of the TransferRequest DTO.

**Listing 13.8: The TransferRequest data transfer object modeling the HTTP request body**

public class TransferRequest {

private long senderAccountId;

private long receiverAccountId;

private BigDecimal amount;

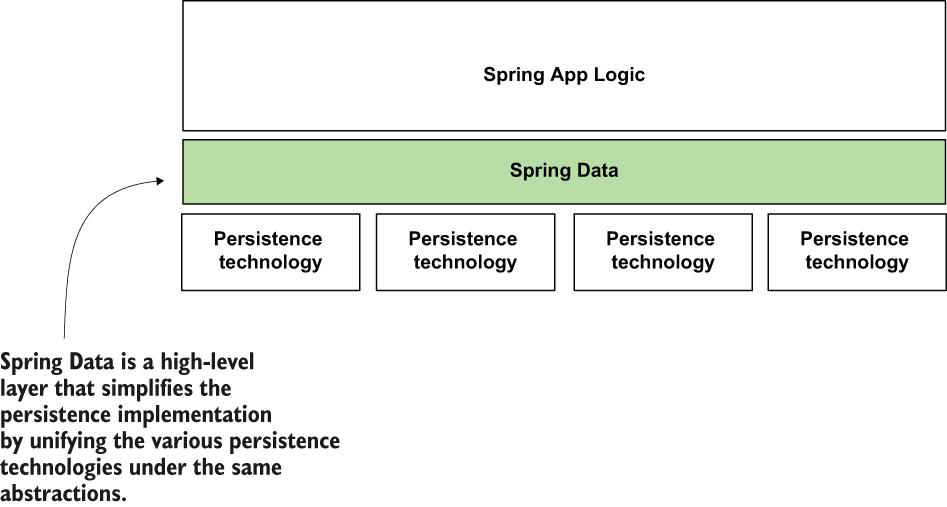
// Omitted code

}

**What is Spring Data?**

Spring Data is a Spring ecosystem project that simplifies the persistence layer's development by providing implementations according to the persistence technology we use.

The Java ecosystem offers a large number of various persistence technologies. You use each technology in a specific way. Each technology has its own abstractions and class design. Spring Data offers a common abstraction layer over all these persistence technologies to simplify the use of multiple persistence technologies.



**What are options available when you are working with JDBC in your Spring app?**

1. You can directly work with JDK interfaces (Statement, PreparedStatement, ResultSet, and so on).
2. you can use JdbcTemplate provide by Spring.

**What are different ways to persist data?**

Using JDBC to connect to a relational DBMS is not the only choice for implementing an app's persistence layer. In real-world scenarios, you'll use other choices as well, and each way to persist data has its own library and set of APIs you need to learn to use. This variety adds a lot of complexity.

A screenshot of a computer

Description automatically generated

The diagram gets more complicated if we include ORM frameworks such as Hibernate.

**Where does hibernate fits in when working with databases for persisting and retrieving data?**

Below diagram show Hibernate's place in the scene.

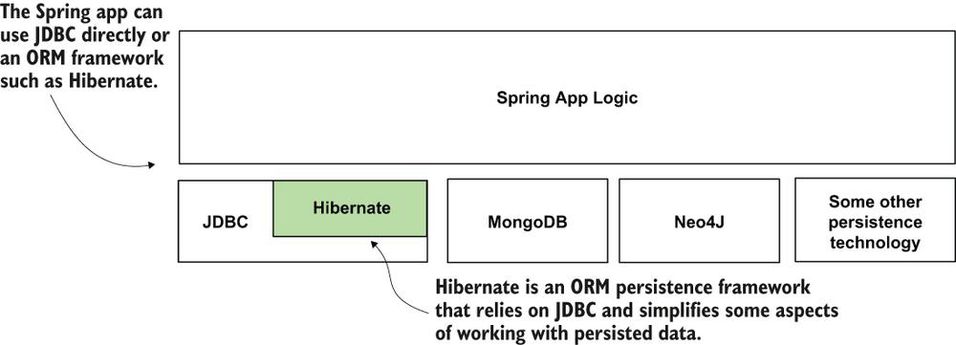
Sometimes apps use frameworks built on top of JDBC, such as Hibernate. The variety in choices makes implementing a persistence layer complex. We want to eliminate this complexity from our apps, and, as you'll learn, Spring Data helps us do this.

A green and black rectangular object with text

Description automatically generated

**Why Spring Data is so important to learning?**

Sometimes apps use frameworks built on top of JDBC, such as Hibernate. The variety in choices makes implementing a persistence layer complex. We want to eliminate this complexity from our apps, and, as you'll learn, Spring Data helps us do this.



**What problems Spring data solves?**

Spring Data simplifies the implementation of the persistence layer by doing the following:

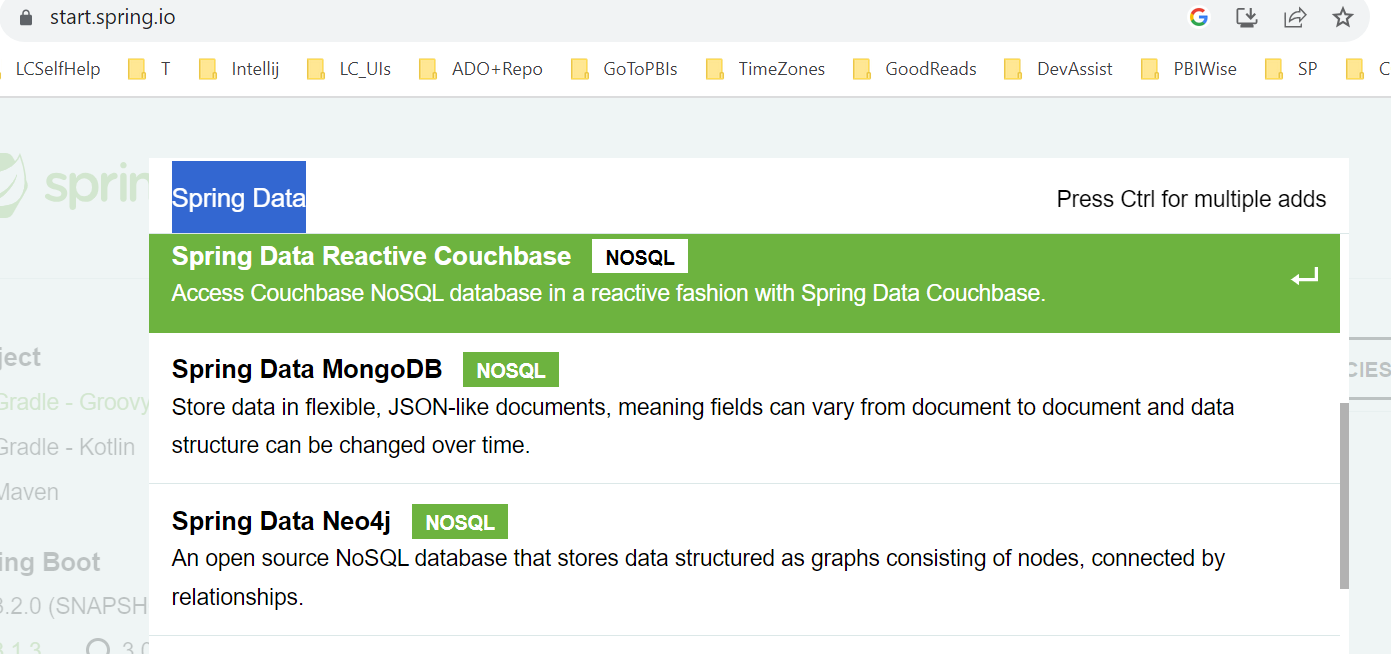
* Providing a common set of abstractions (interfaces) for various persistence technologies. This way, you use a similar approach for implementing the persistence for different technologies.
* Allowing the user to implement the persistence operations using only the abstractions, for which Spring Data provides the implementations. This way, you write less code, so you more quickly implement the app's capabilities. With less written code, the app also becomes easier to understand and maintain.

**What does Spring Data fits in a Spring app?**

Spring Data simplifies the persistence layer implementation by offering a common set of abstractions for various technologies.

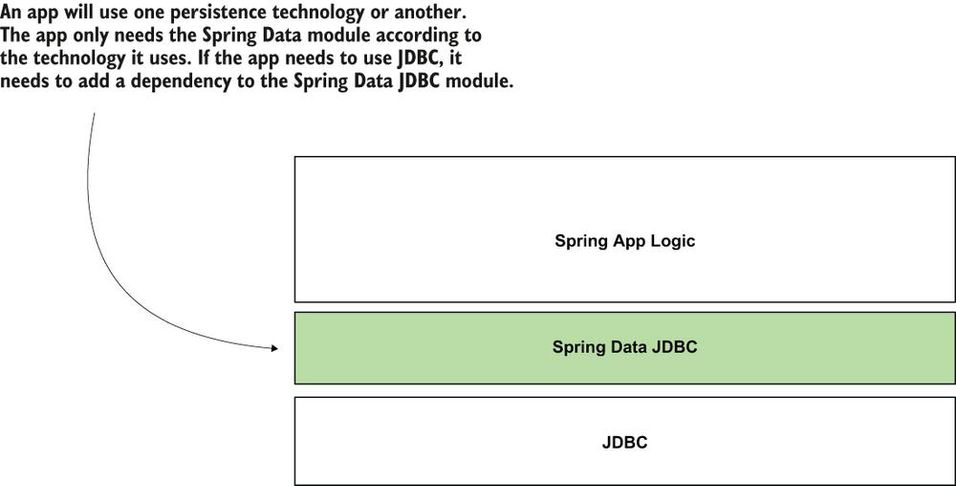
A screenshot of a computer

Description automatically generated



**Which module of Spring data to add if you are using JDBC in your application?**

If the app uses JDBC, it only needs the part of the Spring Data project that manages persistence through JDBC. The Spring Data module that manages the persistence through JDBC is called Spring Data JDBC. You add this Spring Data module to your app through its own dependency.



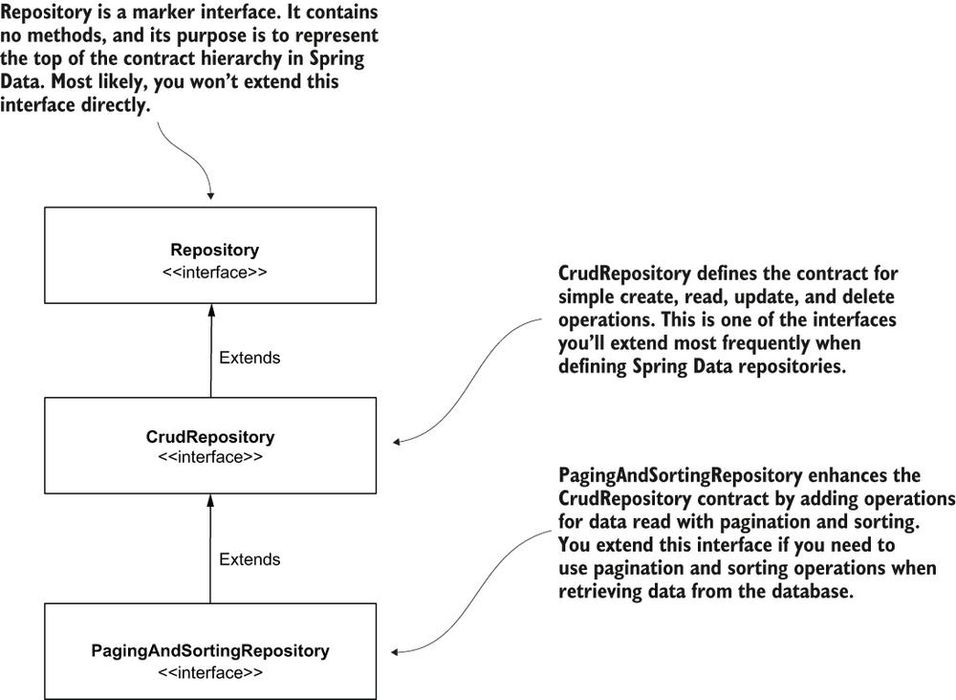
You can find the full list of Spring Data modules on Spring Data's official page: <https://spring.io/projects/spring-data>.

**What are some common set of interface provided by Spring Data?**

 Spring Data provides a common set of interfaces (contracts) you extend to define the app's persistence capabilities. [Figure 14.6](https://cdn2.percipio.com/1694163472.a75dd1d2e106a275a1f6f2de1a7007df35b1b15c/eod/books/158139/OEBPS/section-142-79.xhtml#ch14fig06) presents the following interfaces:

* Repository is the most abstract contract. If you extend this contract, your app recognizes the interface you write as a particular Spring Data repository. Still, you won't inherit any predefined operations (such as adding a new record, retrieving all the records, or getting a record by its primary key). The Repository interface doesn't declare any method (it is a marker interface).
* CrudRepository is the simplest Spring Data contract that also provides some persistence capabilities. If you extend this contract to define your app's persistence capabilities, you get the simplest operations for creating, retrieving, updating, and deleting records.
* PagingAndSortingRepository extends CrudRepository and adds operations related to sorting the records or retrieving them in chunks of a specific number (pages).

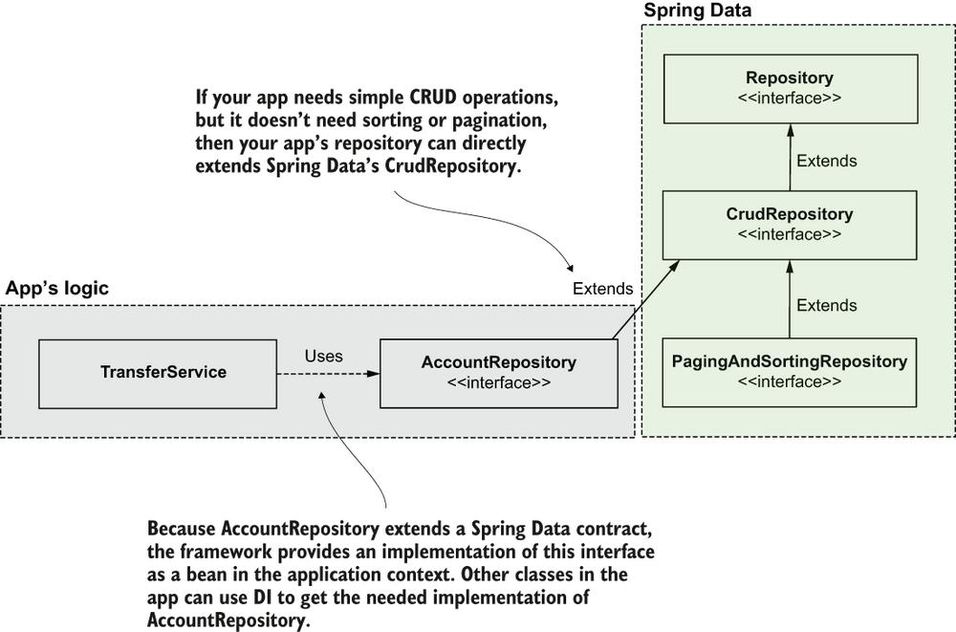
To implement your app's repositories using Spring Data, you extend specific interfaces. The main interfaces that represent Spring Data contracts are Repository, CrudRepository, and PagingAndSortingRepository. You extend one of these contracts to implement your app's persistence capabilities.



**Why Spring Data provides multiple interfaces that extend one another. Why not only one interface with all the operations in it?**

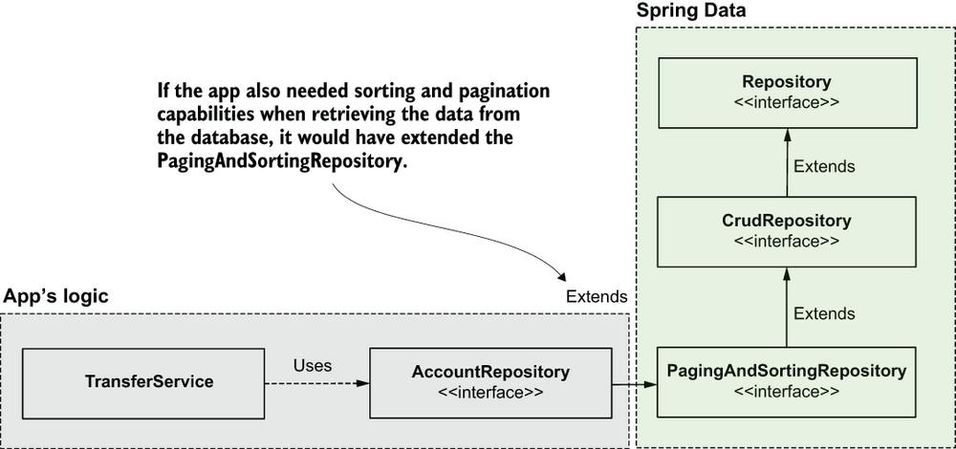
By implementing multiple contracts that extend each other instead of providing you one “fat” contract with all the operations, Spring Data gives your app the possibility to implement only the operations it needs. This approach is a known principle called interface segregation. For example, if your app only needs to use CRUD operations, it extends the CrudRepository contract. Your app won't get the operations related to sorting and paging records, making your app simpler ([figure 14.7](https://cdn2.percipio.com/1694163472.a75dd1d2e106a275a1f6f2de1a7007df35b1b15c/eod/books/158139/OEBPS/section-142-79.xhtml#ch14fig07)).

Figure 14.7: To create a Spring Data repository, you define an interface that extends one of the Spring Data contracts. For example, if your app only needs CRUD operations, the interface you define as a repository should extend the CrudRepository interface. The app adds a bean that implements the contract you define to the Spring context, so any other app components that need to use it can simply inject it from the context.



**When it is better to extend PagingAndSortingRepository** **interface?**

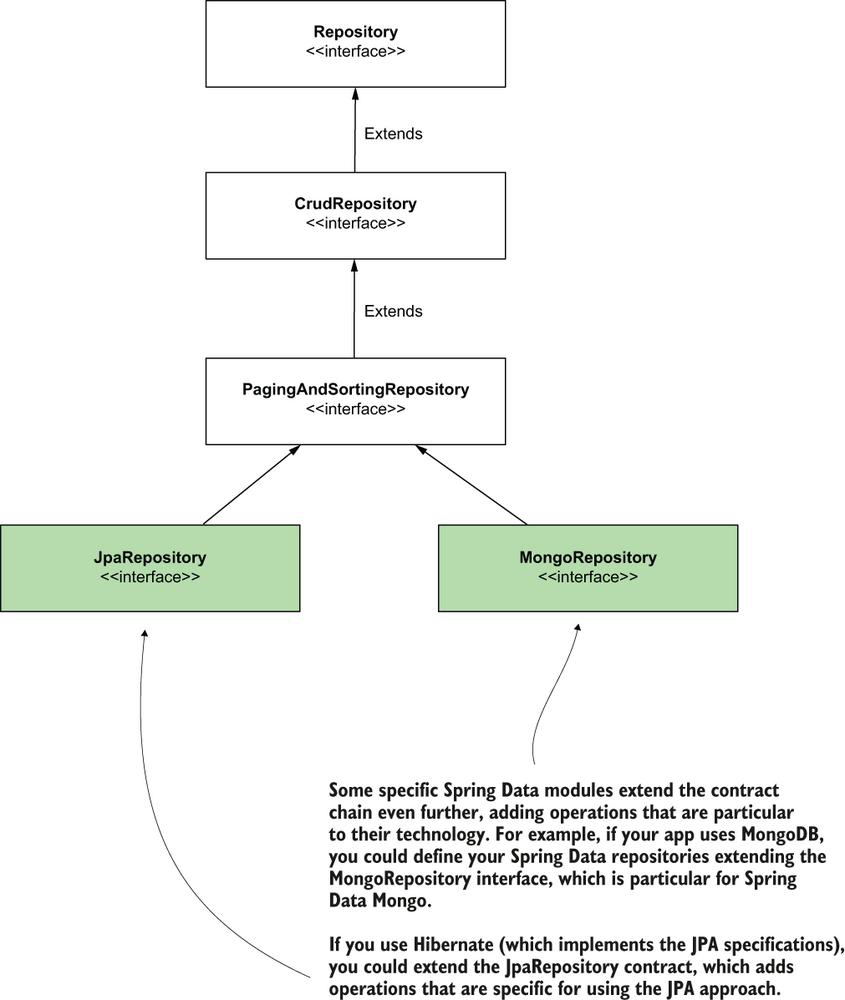
If the app needs sorting and paging capabilities, it should extend a more particular contract. The app provides a bean that implements the contract, which can then be injected from any other component that needs to use it.



**How Spring Data modules provide specific contract, they represent?**

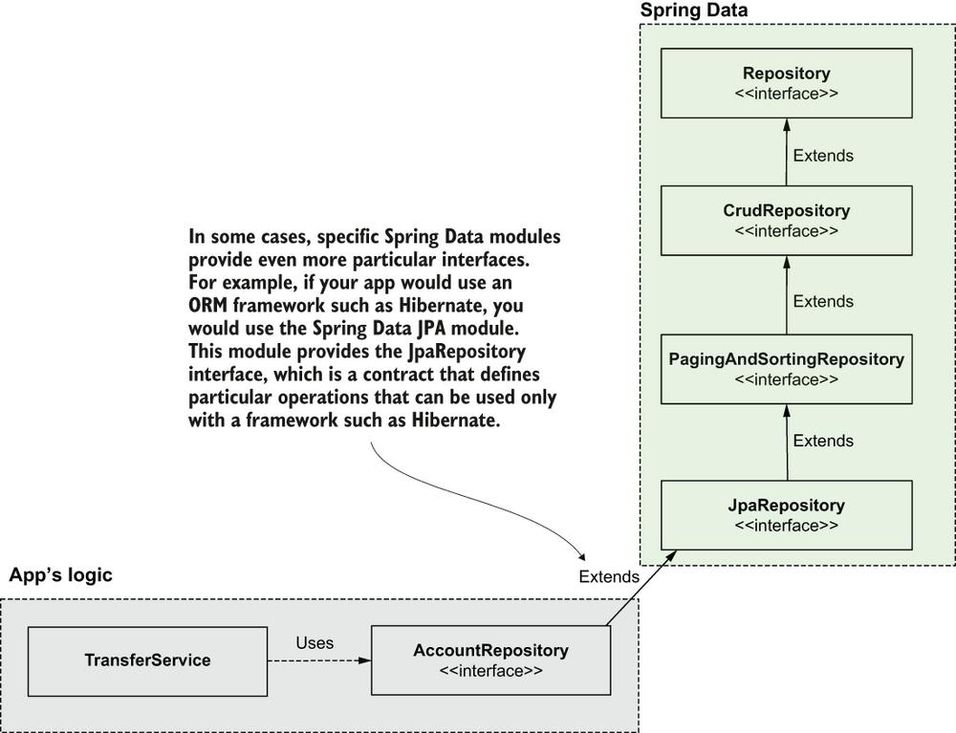
Some Spring Data modules might provide specific contracts to the technology they represent. For example, using Spring Data JPA, you also can extend the JpaRepository interface directly (as presented in [figure 14.9](https://cdn2.percipio.com/1694165312.8c81aa0fa8cf42a2125b6a45263b1882358d2545/eod/books/158139/OEBPS/section-142-79.xhtml#ch14fig09)). The JpaRepository interface is a contract more particular than PagingAndSortingRepository. This contract adds operations applicable only when using specific technologies like Hibernate that implement the Jakarta Persistence API (JPA) specification.

Spring Data modules that are specific to certain technologies might provide particular contracts that define operations you can apply only with those technologies. When using such technologies, your app most likely will use these specific contracts.



**If your project is using Hibernate, then which Spring data module will you use?**

Different Spring Data modules might provide other, more particular contracts. For example, if you use an ORM framework such as Hibernate (which implements the JPA) with Spring Data, you can extend the JpaRepository interface, which is a more particular contract that provides operations applicable only when using a JPA implementation, such as Hibernate.



**Is there any relationship between the repository's method name and the query Spring Data creates behind the scenes?**

The relationship between the repository's method name and the query Spring Data creates behind the scenes.

A diagram of a computer code

Description automatically generated with medium confidence

**What are main disadvantages of relying on the method's name in Spring Data?**

* If the operation requires a more complex query, the method's name would be too large and difficult to read.
* If a developer refactors the method's name by mistake, they might affect the app's behavior without realizing it (unfortunately, not all apps are roughly tested, and we need to consider this).
* Unless you have an IDE that offers you hints while writing the method's name, you need to learn the Spring Data's naming rules. Since you already know SQL, learning a set of rules applicable only for Spring Data is not advantageous.
* Performance is affected because Spring Data also has to translate the method name into a query, so the app will initialize slower (the app translates the method names into queries when the app boots).

The simplest way to avoid these problems is using the @Query annotation to specify the SQL query that the app will run when you call that method. When you annotate the method @Query, it's no longer relevant how you name that method. Spring Data will use the query you provide instead of translating the method's name into a query. The behavior also becomes more performant. The following listing shows you how to use the @Query annotation.

**Listing 14.4: Using the @Query annotation to specify the SQL query for an operation**

public interface AccountRepository

extends CrudRepository<Account, Long> {

@Query("SELECT \* FROM account WHERE name = :name") ❶

List<Account> findAccountsByName(String name);

}

❶ Remember that the parameter's name in the query should be the same as the method parameter's name. There shouldn't be any spaces between the colon (:) and the parameter's name.

However, when your query changes data, you also need to annotate the method with the @Modifying annotation. If you use UPDATE, INSERT, or DELETE, you also need to annotate the method with @Modifying.

public interface AccountRepository

extends CrudRepository<Account, Long> {

@Query("SELECT \* FROM account WHERE name = :name")

List<Account> findAccountsByName(String name);

@Modifying ❶

@Query("UPDATE account SET amount = :amount WHERE id = :id")

void changeAmount(long id, BigDecimal amount);

}

❶ We annotate the methods that define operations that change data with the @Modifying annotation.

**In Spring data, you only create repository interface, how does Spring create a bean out of repository implementation?**

Use DI to get a bean that implements the AccountRepository interface wherever you need it in the app. Don't worry that you only wrote the interface. Spring Data creates a dynamic implementation and adds a bean to your app's context. [Listing 14.6](https://cdn2.percipio.com/1694188119.59b04f56cc06b7903eda19a0de74a55b248c23c3/eod/books/158139/OEBPS/section-143-80.xhtml#ch14list06) shows how the TransferService component of the app uses constructor injection to get a bean of type AccountRepository. In [chapter 5](https://cdn2.percipio.com/1694188119.59b04f56cc06b7903eda19a0de74a55b248c23c3/eod/books/158139/OEBPS/chapter-5-32.xhtml#ch05) you learned that Spring is smart and knows that if you requested a DI for a field with an interface type, it needs to find a bean that implements that interface.

**Listing 14.6: Injecting the repository in the service class to implement the use case**

@Service

public class TransferService {

private final AccountRepository accountRepository;

public TransferService(AccountRepository accountRepository) {

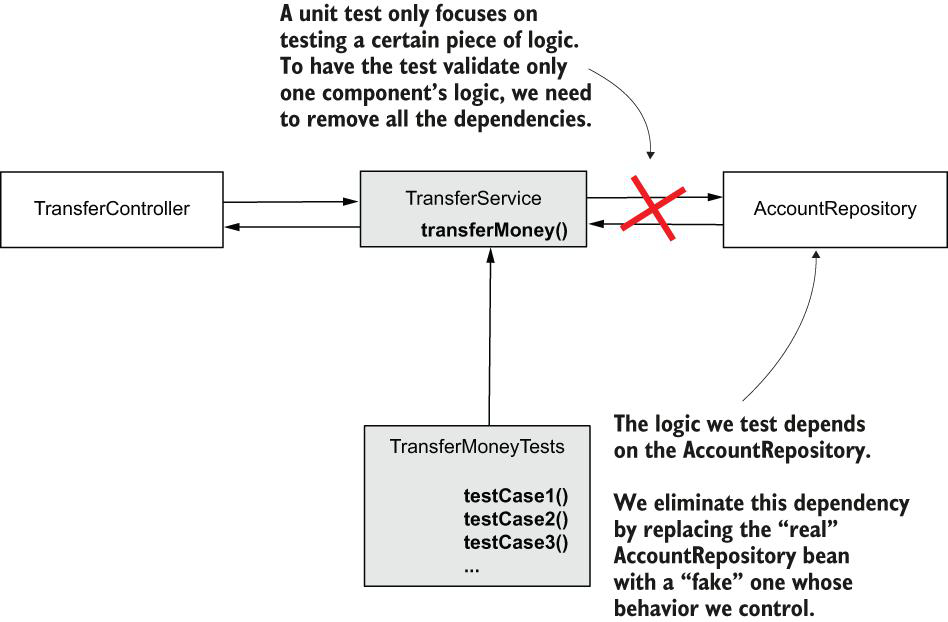
this.accountRepository = accountRepository;

}

}

**Can you explain how we write Unit tests?**

o allow the unit test to focus only on the transferMoney() method's logic, we eliminate the dependency to the AccountRepository object. We use a mock object to replace the real AccountRepository instance, and we control this fake instance to test how the transferMoney() method behaves in different situations



For the examples in this book, we use JUnit 5 Jupiter, the latest JUnit version, to implement the unit and integration tests. However, in real-world apps, you might also find JUnit 4 used often. This is one more reason I recommend you also read books that focus on testing. [Chapter 4](https://cdn2.percipio.com/1694263808.7ad9b181cdad706d97517a34f866448ee8bef4d7/eod/books/158139/OEBPS/chapter-4-27.xhtml#ch04) of JUnit in Action (Manning, 2020) by Cătălin Tudose focuses on the differences between JUnit 4 and JUnit 5.

public class TransferServiceUnitTests {

@Test

public void moneyTransferHappyFlow() {

AccountRepository accountRepository =

mock(AccountRepository.class); ❶

TransferService transferService = ❷

new TransferService(accountRepository);

}

}

❶ We use the Mockito mock() method to create a mock instance for the AccountRepository object.

❷ We create an instance of the TransferService object whose method we want to test. Instead of a real AccountRepository instance, we create the object using a mock AccountRepository. This way, we replace the dependency with something we can control.

**Give a typical example of test case using Mockito?**

@ExtendWith(MockitoExtension.class) ❶

public class TransferServiceWithAnnotationsUnitTests {

@Mock ❷

private AccountRepository accountRepository;

@InjectMocks ❸

private TransferService transferService;

@Test

public void moneyTransferHappyFlow() {

Account sender = new Account();

sender.setId(1);

sender.setAmount(new BigDecimal(1000));

Account destination = new Account();

destination.setId(2);

destination.setAmount(new BigDecimal(1000));

given(accountRepository.findById(sender.getId()))

.willReturn(Optional.of(sender));

given(accountRepository.findById(destination.getId()))

.willReturn(Optional.of(destination));

transferService.transferMoney(1, 2, new BigDecimal(100));

verify(accountRepository)

.changeAmount(1, new BigDecimal(900));

verify(accountRepository)

.changeAmount(2, new BigDecimal(1100));

}

}

❶ Enable the use of @Mock and @InjectMocks annotations.

❷ Use the @Mock annotation to create a mock object and inject it into the test class's annotated field.

❸ Use the @InjectMocks to create the tested object and inject it into the class's annotated field.

Observe how, instead of declaring these objects inside the test method, I took them out as the class parameters and annotated them with @Mock and @InjectMocks. When you use the @Mock annotation, the framework creates and injects a mock object in the annotated attribute. With @InjectMocks annotation, you create the object to test and instruct the framework to inject all the mocks (created with @Mock) in its parameters.

For the @Mock and @InjectMocks annotations to work, you also need to annotate the test class with the @ExtendWith(MockitoExtension.class) annotation. When annotating the class this way, you enable an extension that allows the framework to read the @Mock and @InjectMocks annotations and control the annotated fields.

The main parts of the test implementation. (1) Define and control the dependencies, (2) execute the tested method, and (3) verify the method behaved as expected



**Can you explain the test case for a typical exception scenario?**

@ExtendWith(MockitoExtension.class)

public class TransferServiceWithAnnotationsUnitTests {

@Mock

private AccountRepository accountRepository;

@InjectMocks

private TransferService transferService;

@Test

public void moneyTransferDestinationAccountNotFoundFlow() {

Account sender = new Account();

sender.setId(1);

sender.setAmount(new BigDecimal(1000));

given(accountRepository.findById(1L))

.willReturn(Optional.of(sender));

given(accountRepository.findById(2L))

.willReturn(Optional.empty()); ❶

assertThrows(

AccountNotFoundException.class, ❷

() -> transferService.transferMoney(1, 2, new BigDecimal(100))

);

verify(accountRepository, never()) ❸

.changeAmount(anyLong(), any());

}

}

❶ We control the mock AccountRepository to return an empty Optional when the findById() method is called for the destination account.

❷ We assert that the method throws an AccountNotFoundException in the given scenario.

❸ We use the verify() method with the never() conditional to assert that the changeAmount() method hasn't been called.

**How do test what value a method returns?**

@PostMapping("/")

public String loginPost(

@RequestParam String username,

@RequestParam String password,

Model model

) {

loginProcessor.setUsername(username);

loginProcessor.setPassword(password);

boolean loggedIn = loginProcessor.login();

if (loggedIn) {

model.addAttribute("message", "You are now logged in.");

} else {

model.addAttribute("message", "Login failed!");

}

return "login.html";

}

**Success login scenario**

@ExtendWith(MockitoExtension.class)

class LoginControllerUnitTests {

@Mock

private Model model; ❶

@Mock

private LoginProcessor loginProcessor; ❶

@InjectMocks

private LoginController loginController; ❶

@Test

public void loginPostLoginSucceedsTest() {

given(loginProcessor.login()) ❷

.willReturn(true);

String result = ❸

loginController.loginPost("username", "password", model);

assertEquals("login.html", result); ❹

verify(model) ❺

.addAttribute("message", "You are now logged in.");

}

}

❶ We define the mock objects and inject them into the instance whose behavior we test.

❷ We control the LoginProcessor mock instance, telling it to return true when its method login() is called.

❸ We call the tested method with the given assumptions.

❹ We verify the tested method returned value.

❺ We verify the message attribute was added with the correct value on the model object.

**Failed login scenario?**

@ExtendWith(MockitoExtension.class)

class LoginControllerUnitTests {

// Omitted code

@Test

public void loginPostLoginFailsTest() {

given(loginProcessor.login())

.willReturn(false);

String result =

loginController.loginPost("username", "password", model);

assertEquals("login.html", result);

verify(model)

.addAttribute("message", "Login failed!");

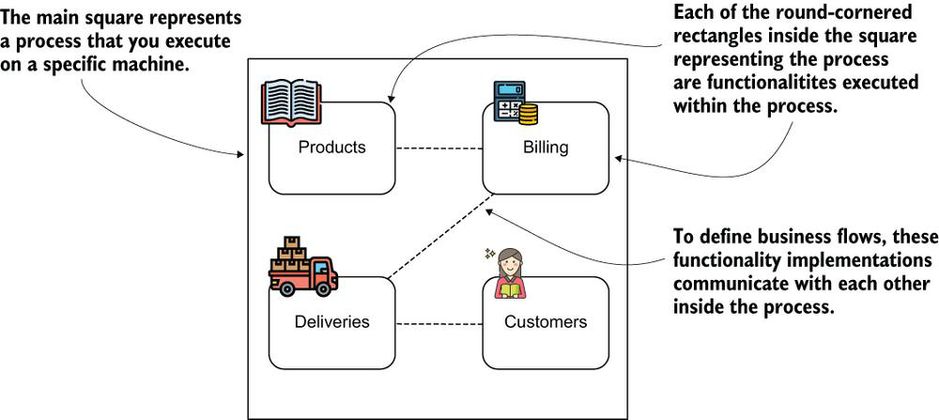
}

}

**What is Monolith?**

When developers refer to an app as being “monolithic” or “a monolith,” it means that it consists of just one component you deploy and execute. This component implements all its functionalities.

A monolithic application. The application implements all the functionalities in just one process. The implementations interact with one another inside the process to develop the business flow.



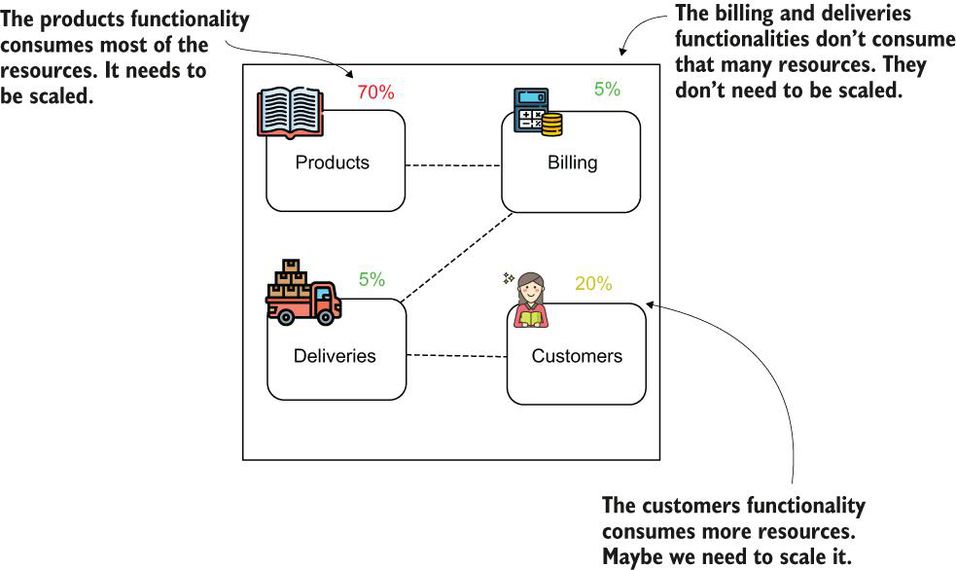
Horizontal scaling. Running the same instance multiple times enables us to use more resources and serve more client requests.

Diagram of a load balancer

Description automatically generated

**What is motivation behind SOA architecture?**

Some features are more intensively used than others. For this reason, these features consume more resources and need to be scaled.



In a SOA, each feature is an independent process. This way, you can decide to scale just the features that need more resources.



A monolithic system consists of only one app, so if you have multiple teams working on the system, they all work on the same app. This approach requires more coordination. In a SOA where the system is composed of multiple apps, each team can work on different apps. This way, they need to coordinate less.



**What are some complexities, you introduce while working on SOA architecture?**

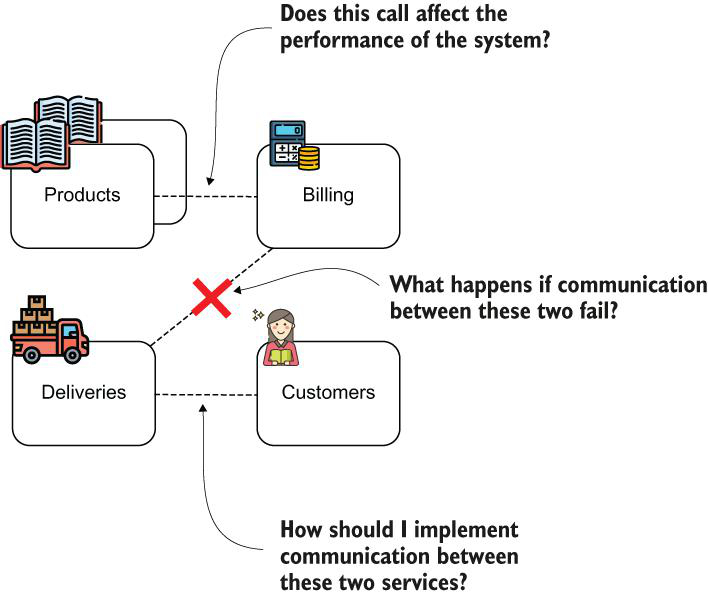
1. Communication among the services
2. Security
3. Data persistence
4. Deployment

**How do explain complexity caused by communication between two services in SOA architecture?**

Features now need to communicate via the network. One of the essential principles you need to remember is that the network isn't entirely reliable. Many fall into the trap of forgetting to consider what happens if, at some point, the communication between two components breaks. Unfortunately, unlike a monolithic approach, any call between two components can fail at some point in a SOA. Depending on the app, developers use different techniques or patterns to solve this issue, like repeating calls, circuit breakers, or caches.

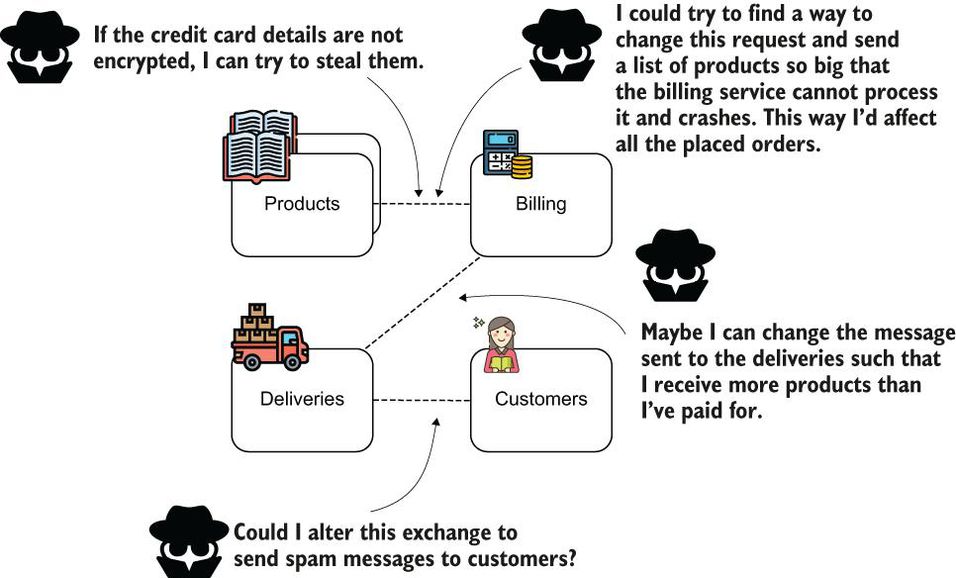
A second aspect to consider is that there are many options to establish communication among the services ([figure A.7](https://cdn2.percipio.com/1694417046.62717f9484f4e1c2473d4cb06a3eac4c82dc3db4/eod/books/158139/OEBPS/section-A2-88.xhtml#app01fig07)). You could use REST services, GraphQL, SOAP, gRPC, JMS message brokers, Kafka, and so on. Which is the best approach? Of course, in any situation one or more of these approaches is fine. You'll find long debates and discussions in many books on how to choose the right fit for typical scenarios.

Communication between services adds complexity to the system. We need to decide how to implement the communication between two services. We also need to understand what could happen if the communication fails and how to solve potential problems caused by malfunctioning communication



**How does complexity is added in SOA architecture in respect to security of the system?**

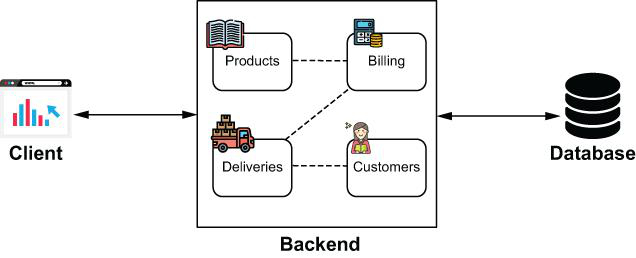
In a SOA, features are separate services and communicate over the network. This aspect introduces many vulnerable points that developers need to consider when building the app.



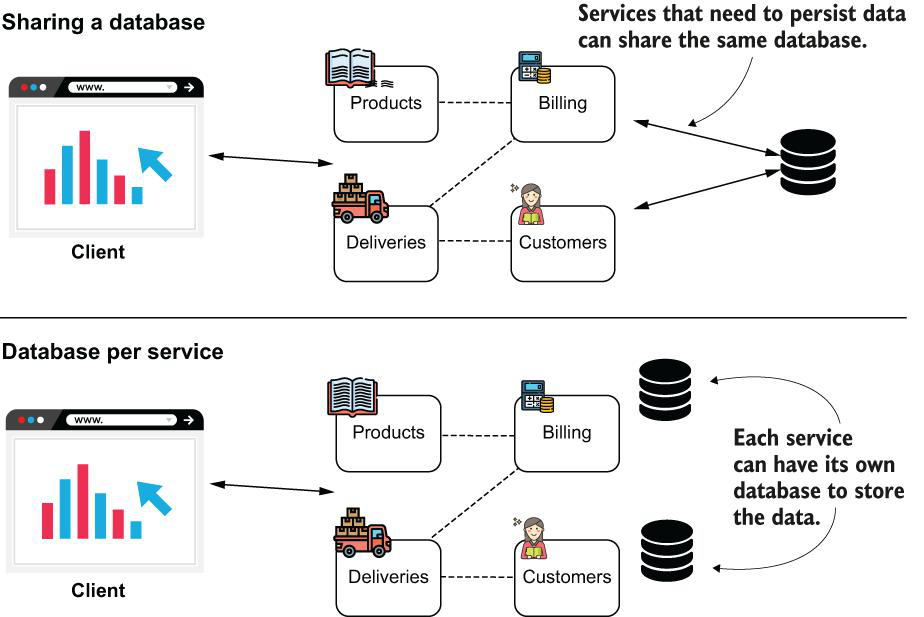
**What is tree-tier architecture?**

With a monolithic approach, an app had one database to store the data, as presented in [figure A.9](https://cdn2.percipio.com/1694417046.62717f9484f4e1c2473d4cb06a3eac4c82dc3db4/eod/books/158139/OEBPS/section-A2-88.xhtml#app01fig09). We called this a three-tier architecture because it consists of three tiers: the client, the backend, and the database used for persistence.

With a monolithic approach, you only have one application and usually one database. The system is simple and can be easily visualized and understood.



In a SOA, you can decide that more services share the same database or have an individual database per service. Having various alternatives, each with its benefits and drawbacks, makes the persistence layer's design in SOA more difficult.



Most believe that sharing a database is bad practice. From my own experience with splitting a monolith into multiple services, I can tell you that having a shared database can become a deployment nightmare. But having individual databases per service also implies difficulties. As you'll see when we discuss transactions, it's much easier to assure data consistency with one database. When having more independent databases, it's challenging to make sure the data remains consistent among all.

**Why does a monolith have a negative connotation?**

You can see that the SOAs aren't necessarily easy, so you might wonder why monolithic architecture tends to be associated with something negative. The reality is that for some systems, a monolith makes more sense than SOA.

My opinion is that the negative connotation of monolithic architecture comes from the fact that it represents old systems. In most cases, old systems were implemented before anyone was concerned about clean coding and design principles. We now consider all these principles to make sure we write maintainable code.

It might feel strange to look back to the times when they didn't exist, and sometimes I've even seen developers blame those who started the implementation of such old systems when problems arise. But the truth is that it's not the fault of the folks who used the tools and practices that everyone considered the best at that time.

Today, many developers associate messy and poorly written code with a monolith concept. However, monolithic apps can be modular, and their code can be clean, while service-oriented apps can be messy and poorly designed.

**What is a microservice?**

Microservices are a particular implementation of the SOA. A microservice usually is designed with one responsibility and has its own persistence capability (it doesn't share databases).

**What is DevOps?**

A wise software architect knows to adapt the system's architecture to both the way teams work on the system and how the system is deployed. You might have heard about what we call the DevOps movement, which implies both how we deploy software as well as how we work on software development.

**How deployment of apps on virtual machines or containerized environments have implied on size of apps?**

We deploy the apps in the cloud using virtual machines or containerized environments, and these approaches generally implied the need for making apps smaller.

Of course, evolution came with another incertitude: how small should a service be? Many debated this question in books, articles, and discussions.

**What are serverless functions?**

The minimization of services went so far that today we can implement a short functionality with only a few lines of code and deploy it in an environment. An event like an HTTP request, a timer, or a message triggers this functionality and makes it execute. We call these small implementations serverless functions. The term “serverless” doesn't imply that the function doesn't execute on a server. But because everything regarding the development is hidden and we only case for the code that implements its logic and the events that trigger it, it merely looks like no server exists.

**What is recommended books to understand software architecture and its evolution?**

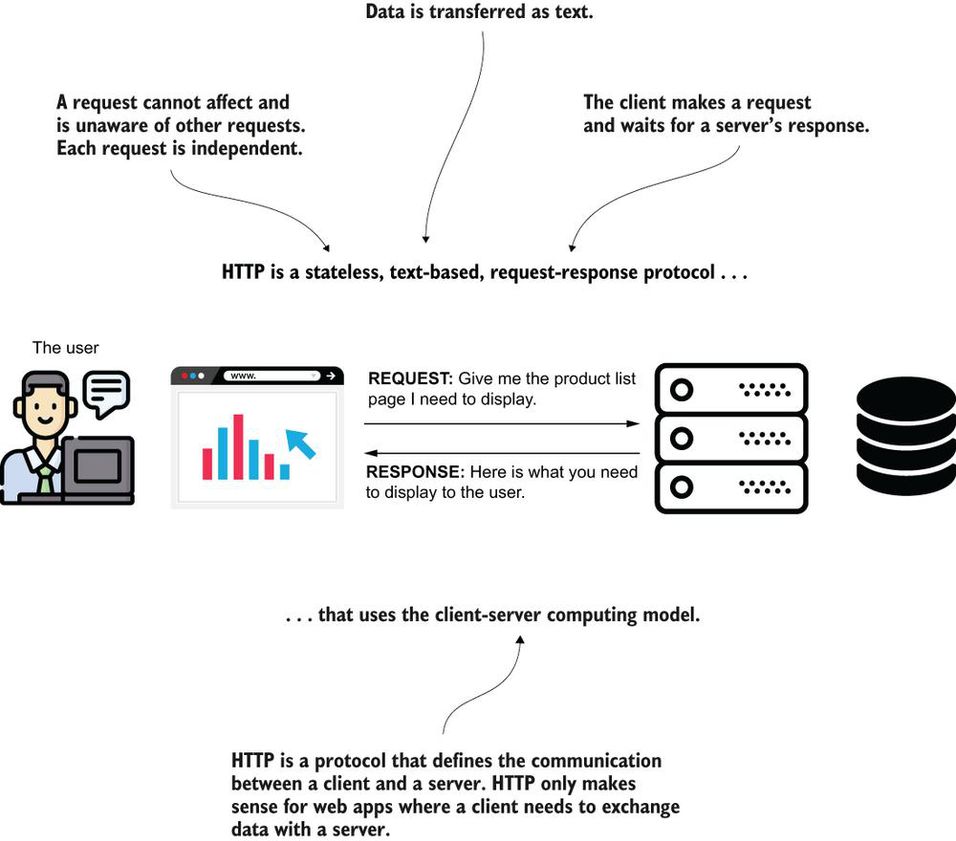
Software architecture and its evolution is such a fantastic and complex subject. I don't think there'll ever be too many books to cover this subject thoroughly. I've added this discussion to the book to help you understand the references I'll make to these notions. Still, you might want to go deeper into the subjects, so here's a list of books from my shelf. The books are in the order I recommend you read them.

1. Microservices in Action, by Morgan Bruce and Paulo A. Pereira (Manning, 2018), is an excellent book you can start with when learning microservices. In the book, you'll find all the microservices fundamentals subjects discussed with useful examples.
2. Microservices Patterns, by Chris Richardson (Manning, 2018), is a book I recommend you continue with after thoroughly reading Microservices in Action. The author presents a pragmatic approach on how to develop production-ready apps using microservices.
3. Spring Microservices in Action, by John Carnell and Illary Huaylupo Sánchez (Manning, 2020), helps you better understand how to apply Spring to build microservices.
4. Microservices Security in Action, by Prabath Siriwardena and Nuwan Dias (Manning 2020), goes into detail with what applying security with a microservice architecture means. Security is a crucial aspect of any system, and you always need to consider it from the development process's early stages. The book explains security from the ground up, and reading it will give you a better understanding the aspects you need to be aware of in regards to security for microservices.
5. Monolith to Microservices, by Sam Newman (O'Reilly Media, 2020), treats patterns for transforming a monolithic architecture into microservices. The book also discusses whether you need to use microservices and how to decide this.

**What is HTTP?**

Stateless, text-based, request-response protocol that uses the client-server computing model.

HTTP is a protocol that describes how a client and a server talk. HTTP assumes a client makes a request, and the server responds. The protocol describes what the client's request and the server's response look like. HTTP is stateless, meaning the requests are independent of one another, and text-based, which means the information is exchanged as plain text

Larger View

**What are components of an HTTP request?**

The HTTP request has a simple format. The things you have to take into consideration are the following:

1. The request URI—The client uses the path to tell the server what resource it requests. The request URI looks like this one: <http://www.manning.com/books/spring-start-here>
2. The request method—A verb that the client uses to indicate what action it will do with the requested resource. For example, when you write an address in a web browser's address bar, the browser always uses an HTTP method named GET. In other circumstances, you'll find in the next paragraphs, the client can issue an HTTP request with a different method such as POST, PUT, or DELETE.
3. The request parameters (optional)—Data in small quantity the client sends to the server with the request. When I say “small quantity,” I refer to something that can be expressed in maybe 10 to 50 characters. Parameters on the request aren't mandatory. The request parameters (also referred to as query parameters) are sent in the URI by appending a query expression.
4. The request headers (optional)—Data in small quantity sent in the request header. Unlike request parameters, these values are not visible in the URI.
5. The request body (optional)—A larger quantity of data the client sends to the server in the request. When the client needs to send data composed of some hundreds of characters, it can use the HTTP body. A body on the request is not mandatory.

**What are different HTTP methods?**

The HTTP methods you'll find most often in web apps are as follows:

* GET—Expresses the client's intention to obtain some data from the server.
* POST—Expresses the client's intention to add data on the server.
* PUT—Expresses the client's intention to change data on the server.
* DELETE—Expresses the client's intention to remove some data from the server.

**What is convention behind use relevant HTTP methods?**

Always remember that the verbs are not a constraint to what you implement. The HTTP protocol cannot force you not to implement an HTTP GET functionality that changes data on the backend side. However, you should never misuse the HTTP methods! Always consider the meaning of the HTTP method used to ensure your app's reliability, security, and maintainability.

**What are different options to send data from client to server?**

1.URI path Path http://\*\*\*/books/1

2.Request parameters(query parameters), http://\*\*\*?page=1

3.Request headers,

4.Request body

**Can you explain HTTP response?**

HTTP is the protocol that allows the client to communicate with the server in a web app. Once you take care of the client's request in an app, it's time to implement the server's response. In response to a client's request, the server sends the following:

* The response status—An integer between 100 and 599 that defines a short representation of the request's result.
* Response headers (optional)—Similar to request parameters, they represent key-value pair data. They are designed for sending a small amount of data (10 to 50 characters) from server to client in response to a client's request.
* The response body (optional)—A way for the server to send a larger quantity (e.g., the server needs to send some hundreds of characters or entire files) of data back to the client.

The response status is the only mandatory detail a server delivers in response to a client's request. The status tells the client if the server understood the request and everything worked fine, or if the server encountered issues while processing the client's request.

**What are some important HTTP response status code?**

ou don't need to learn all the statuses in detail. I'll enumerate and describe the ones you'll find more often in real-world implementations:

* Starting with 2, meaning the server correctly processed the request. The request processing is okay, and the server executed what the client asked.
* Starting with 4, where the server tells the client something is wrong with its request (it's a problem on the client side). For example, the client requested a resource that doesn't exist, or the client sent some request parameters that the server didn't expect.
* Starting with 5, where the server communicates that something went wrong on its side. For example, the server needed to connect to a database but the it was not accessible. In this case, the server sends back a status telling the client that it couldn't complete the request but not because of something the client didn't do well.

Different values starting with 2 are variations of messages saying that the server correctly processed the client's request. A few examples are as follows:

* 200—OK is the most known and most straightforward of the response statuses. It just tells the client the server didn't encounter any issue when processing its request.
* 201—Created might be used, for example, in response to a POST request to tell the client that the server managed to add the requested resource. It's not always mandatory to add such detail to the response status, and that's why 200—OK is, in general, the most used response status to identify that everything's okay.
* 204—No Content could tell the client it shouldn't expect a response body for this response.

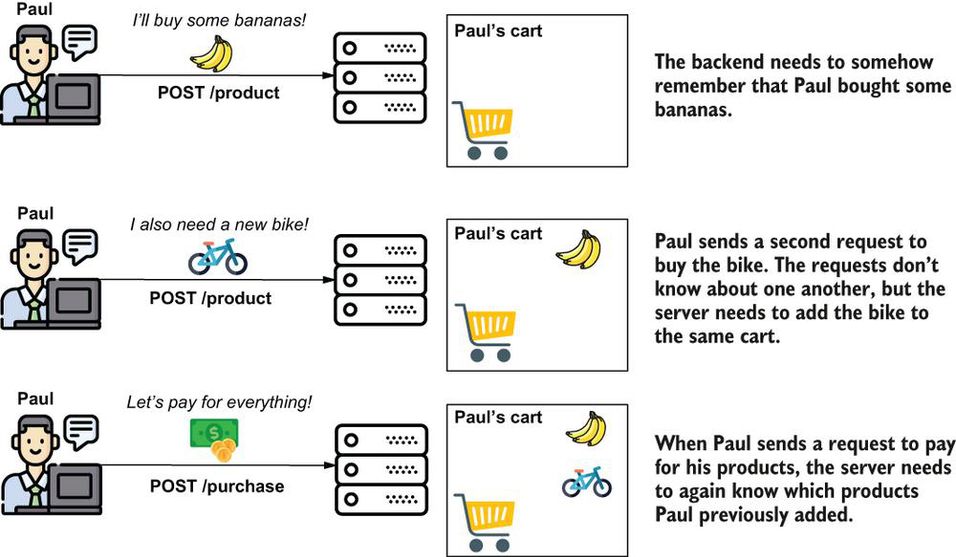
When an HTTP response status value starts with 4, the server tells the client something was wrong with the request. The client did something wrong when requesting a specific resource. It could be that the resource doesn't exist (the well-known 404—Not Found), or maybe some validation of the data didn't go well. Some of the most often encountered client error response statuses are as follows:

* 400—Bad Request—A generic status often used to represent any kind of problem with the HTTP request (e.g., validation of the data or problem with reading a specific value in the request body or a request parameter).
* 401—Unauthorized—A status value generally used to communicate to the client that the request needs authentication.
* 403—Forbidden—A status value generally sent by the server to tell the client it's not authorized to execute its request.
* 404—Not Found—A status value sent by the server to inform the client the requested resource doesn't exist.

When the response status starts with 5, it means something went wrong on the server side, but it's the server's issue. The client sent a valid request, but the server could not complete it for some reason. The most often used status from this category is 500—Internal Server Error. This response status is a generic error value the server sends to inform the client that an issue occurred while the backend was processing its request.

**Where do HTTP session come into play in web applications?**

For an online shop, the backend needs to identify the clients and remember the products they added to their carts. The HTTP requests are independent one from another, so the backend needs to find another way to remember the products added by each client.



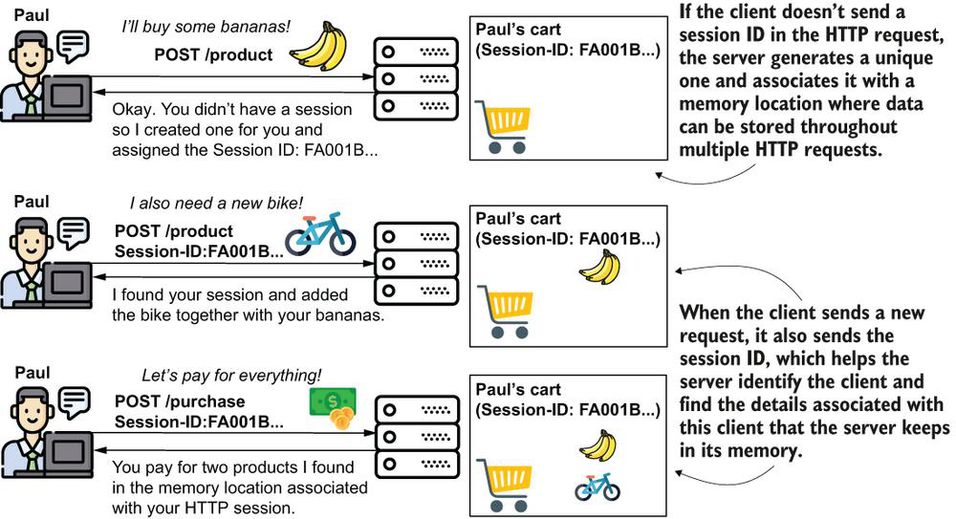
One way to implement such behavior is using the HTTP session. The backend assigns a unique identifier named “session ID” to a client and then associates it with a place in the app's memory. Each request the client sends after being assigned the session ID needs to contain the session ID in a request header. This way, the backend app knows to associate the specific session requests.

**How is session management happening in your current project?**

Using external Redis Cache.

**How does session management works. Can you explain with a typical example.**

The HTTP session mechanism. The server identifies the client with a unique session ID it generates. The client sends the session ID in the next requests, so the backend app knows which memory location it reserved earlier for the client



**How should session expiration time be configured?**

The HTTP session usually ends after a time if the client doesn't send more requests. You can configure this time, usually both in the servlet container and the app. It shouldn't be more than maybe a few hours. If the session lives too long, the server will spend a lot of memory. For most apps, a session ends after less than one hour if the client doesn't send more requests.

If the client sends another request after the session ended, the server will start a new session for that client.

**What are microservices?**

Microservices are distributed, loosely coupled software services that carry out a small number of well-defined tasks.

**How deployment work in Monolithic architecture?**

In a monolithic architecture, an application is delivered as a single deployable software artifact. All the UI (user interface), business, and database access logic are packaged together into a single application artifact and deployed to an application server.

 A microservice is a small, loosely coupled, distributed service. Microservices allow you to take a large application and decompose it into easy-to--manage components with narrowly defined responsibilities. Microservices help combat the traditional problems of complexity in a large code base by decomposing the large code base down into small, well-defined pieces. The key concept you need to embrace as you think about microservices is decomposing and unbundling the functionality of your applications so they’re completely independent of one another.

**What are different platform types?**

IaaS,

PaaS,

SaaS,

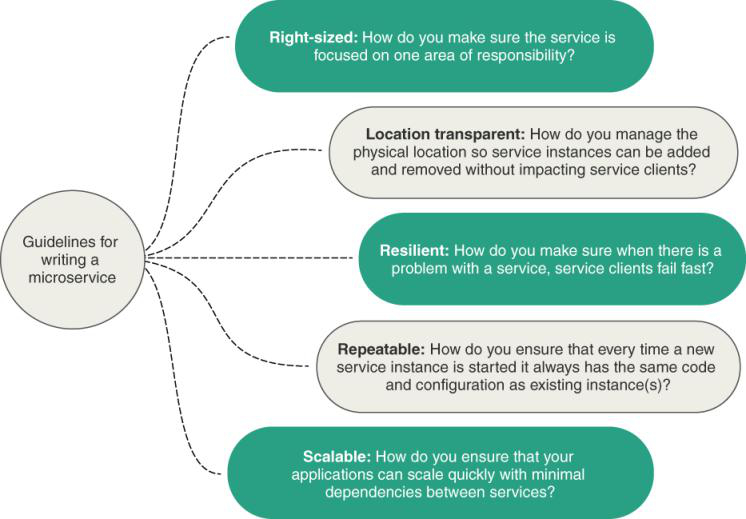
FaaS (Function as a Service), ex. Google Cloud function

CaaS (Container as a Service), Docker

## **How do you explain “Microservices are More than Writing the Code”?**

While the concepts around building individual microservices are easy to understand, running and supporting a robust microservice application (especially when running in the cloud) involves more than writing the code for the service. Writing a robust service includes considering several topics. [Figure 1.7](https://cdn2.percipio.com/1694712553.e12f530fba1ad7cbe1ee88d9c7795a7260200219/eod/books/147121/OEBPS/section-19-16.xhtml#ch01fig07) highlights these topics.

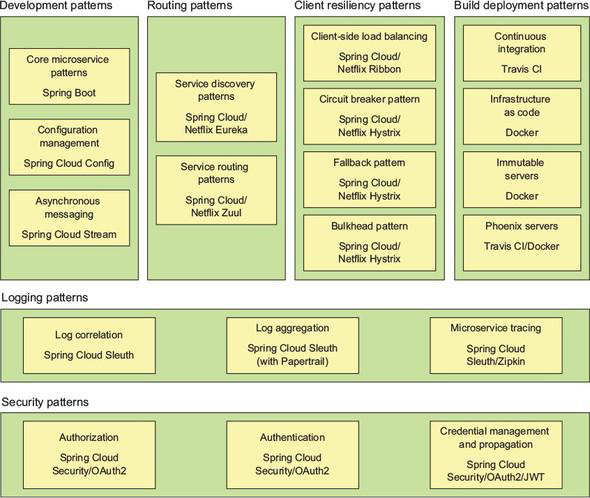
Microservices are more than the business logic. You need to think about the environment where you’ll run the services and how the services will scale and be resilient



**What is Spring cloud?**

Spring Cloud wraps the work of open source companies such as Pivotal, HashiCorp, and Netflix in delivering patterns. Spring Cloud simplifies setting up and configuring of these projects into your Spring application so that you can focus on writing code, not getting buried in the details of configuring all the infrastructure that can go with building and deploying a microservice application.

You can map the technologies you’re going to use directly to the microservice patterns we’ve explored so far in this chapter.



**When not to use microservices?**

### 2.2.1 Complexity of Building Distributed Systems

Because microservices are distributed and fine-grained (small), they introduce a level of complexity into your application that wouldn’t be there in more monolithic applications. Microservice architectures require a high degree of operational maturity. Don’t consider using microservices unless your organization is willing to invest in the automation and operational work (monitoring, scaling) that a highly distributed application needs to be successful.

### 2.2.2 Server Sprawl

One of the most common deployment models for microservices is to have one microservice instance deployed on one server. In a large microservices-based application, you might end up with 50 to 100 servers or containers (usually virtual) that have to be built and maintained in production alone. Even with the lower cost of running these services in the cloud, the operational complexity of having to manage and monitor these servers can be tremendous.

The flexibility of microservices has to be weighed against the cost of running all of these servers.

### 2.2.3 Type of Application

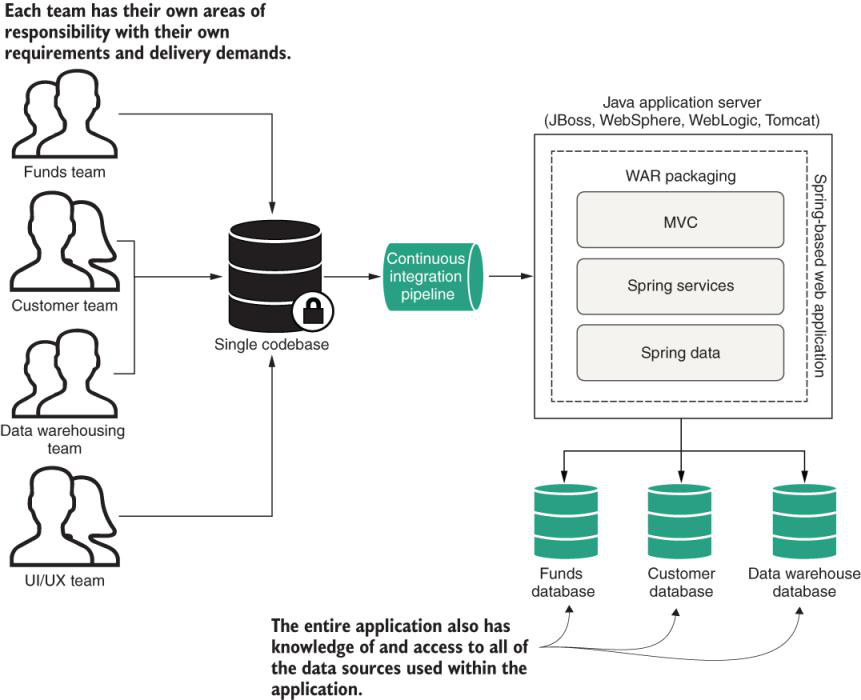
Microservices are geared toward reusability and are extremely useful for building large applications that need to be highly resilient and scalable. This is one of the reasons why so many cloud-based companies have adopted microservices. If you’re building small, departmental-level applications or applications with a small user base, the complexity associated with building on a distributed model such as microservices might be more expense than it’s worth.

### 2.2.4 Data Transformations and Consistency

If your applications need to do complex data aggregation or transformation across multiple sources of data, the distributed nature of microservices will make this work difficult. Your microservices will invariably take on too much responsibility and can also become vulnerable to performance problems.

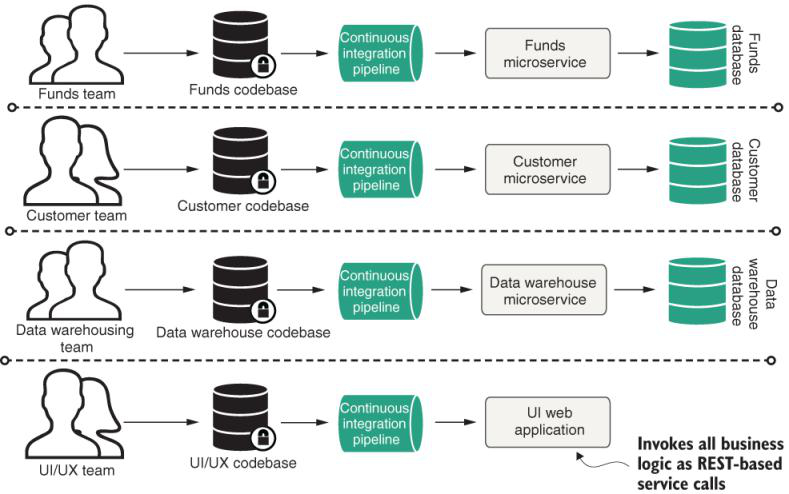
**How do teams collaborate in a typical monolithic application look like?**

Monolithic applications force multiple development teams to synchronize their delivery date because their code needs to be built, tested, and deployed as an entire unit.



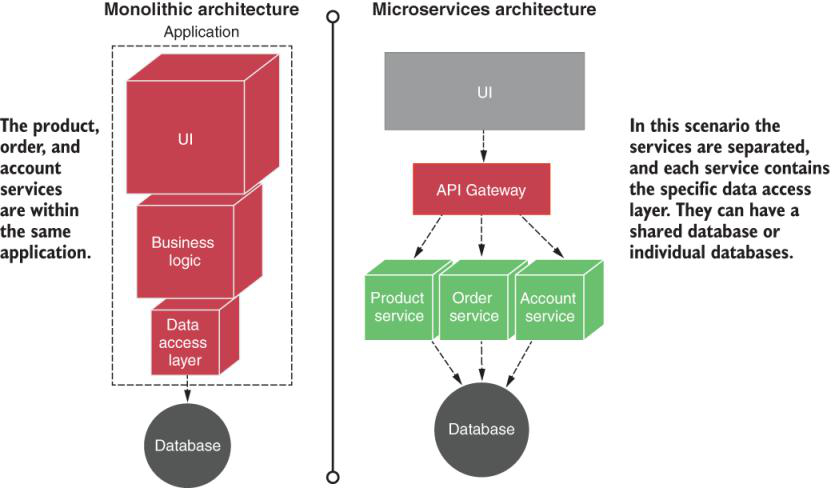
**How do teams collaborate in a typical microservice application look like?**

Using a microservice architecture, a CRM application is decomposed into a set of completely independent microservices, allowing each development team to move at its own pace.



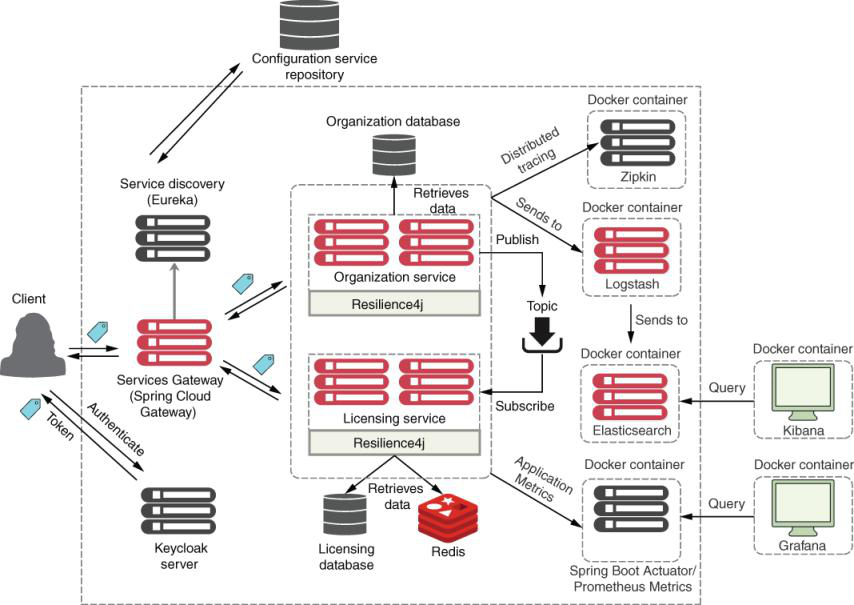
**Can you explain differences between monolithic app and microservices?**

Comparing monolithic and microservices architectures.



**Can you give high level overview of spring cloud components that we use in microservices architecture?**

 High-level overview of the services and technologies.



### What Exactly is Cloud Computing?

### Cloud computing is the delivery of computing and virtualized IT services—databases, networking, software, servers, analytics, and more—through the internet to provide a flexible, secure, and easy-to-use environment. Cloud computing offers significant advantages in the internal management of a company, such as low initial investment, ease of use and maintenance, and scalability, among others.

### The cloud computing models let the user choose the level of control over the information and services that these provide. These models are known by their acronyms, and are generically referred to as XaaS —an acronym that means anything as a service.

IaaS platforms include AWS (EC2), Azure Virtual Machines, Google Compute Engine, and Kubernetes.

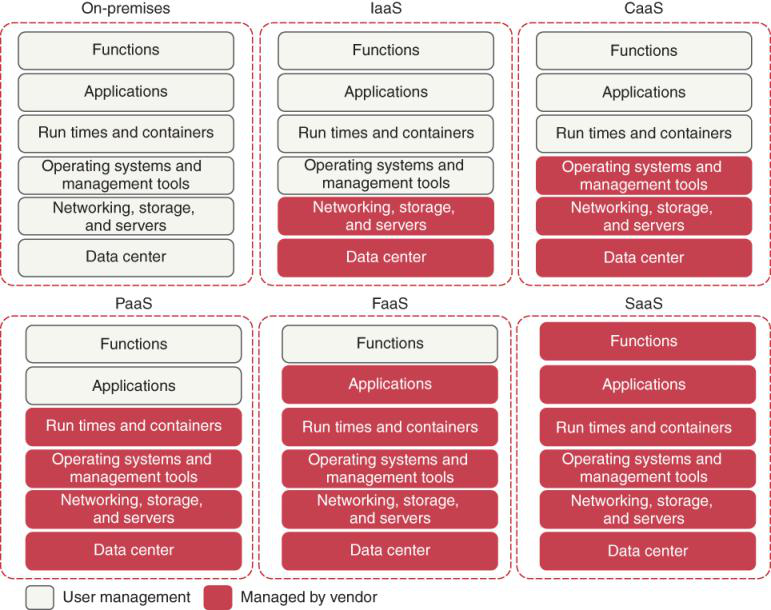
CaaS platforms include Google Container Engine and Amazon’s Elastic Container Service (ECS). In [chapter 11](https://cdn2.percipio.com/1694774644.38bc2322877e895203b6245c83b522586af8006e/eod/books/157553/OEBPS/chapter-11-91.xhtml#ch11), we’ll see how to deploy the microservices you’ve built to Amazon ECS.

PaaS platforms include Google App Engine, Cloud Foundry, Heroku, and AWS Elastic Beanstalk.

FaaS platforms include AWS (Lambda), Google Cloud Function, and Azure functions.

SaaS platforms include Salesforce, SAP, and Google Business.

The different cloud computing models come down to who’s responsible for what: user management or cloud vendor



**What are some cloud based services which allow you to not bother about deployments?**

Cloud Foundry, AWS Elastic Beanstalk, Google App Engine, and Heroku give you the ability to deploy your services without having to know about the underlying application container.

These provide a web interface and command-line interface (CLI) to allow you to deploy your application as a WAR or JAR file. Setting up and tuning the application server and the corresponding Java container are abstracted away from you.

**What are some microservices patterns?**

1.Core development pattern

2.Routing patterns

3.Client resiliency patterns

4.Security patterns

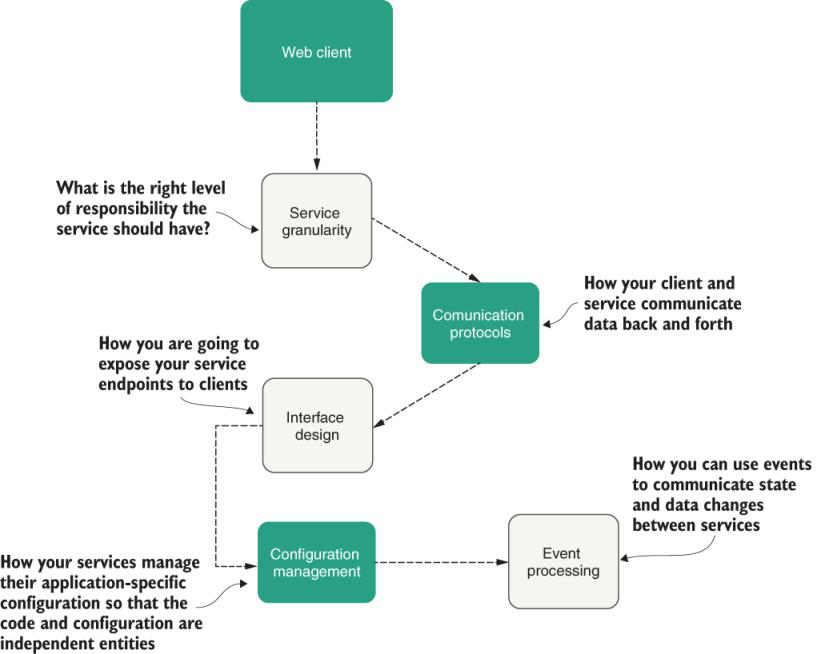
5.Logging and tracing patterns

6.Application metrics patterns

7.Build and deployment patterns.

**What is Core Microservice Development Pattern?**

When designing your microservice, you need to think about how the service will be consumed and communicated with.



Service granularity—How do you approach decomposing a business domain down into microservices so that each microservice has the right level of responsibility?

Communication protocols—How will developers communicate with your service? The first step is to define whether you want a synchronous or asynchronous protocol. For synchronous, the most common communication is HTTP-based REST using XML (Extensible Markup Language), JSON (JavaScript Object Notation), or a binary protocol such as Thrift to send data back and forth to your microservices. For asynchronous, the most popular protocol is AMQP (Advanced Message Queuing Protocol) using a one-to-one (queue) or a one-to-many (topic) with message brokers such as RabbitMQ, Apache Kafka, and Amazon Simple Queue Service (SQS). In later chapters, we’ll learn about the communication protocols.

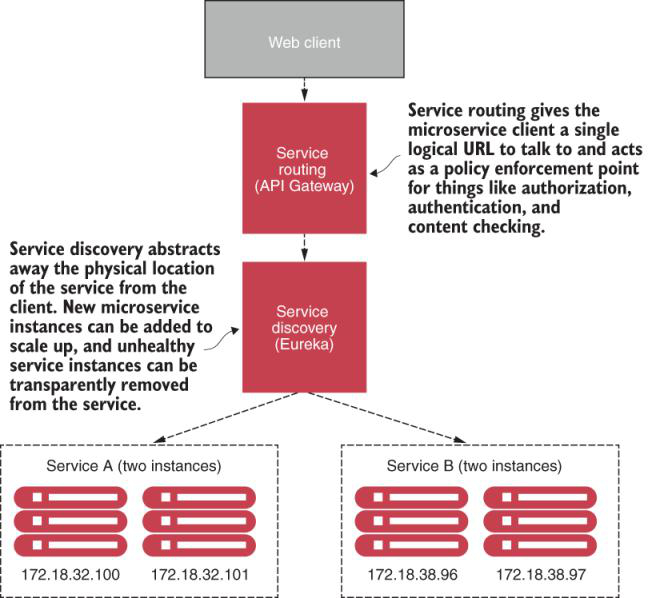
Interface design—What’s the best way to design the actual service interfaces that developers are going to use to call your service?

Configuration management of service—How do you manage the configuration of your microservice so that it moves between different environments in the cloud? This can be managed with externalized configuration and profiles.

Event processing between services—How do you decouple your microservice using events so that you minimize hardcoded dependencies between your services and increase the resiliency of your application? We’ll use an event-driven architecture with Spring Cloud Stream.

**What is microservice routing pattern?**

Service discovery and service routing are key parts of any large-scale microservice application.



**Service discovery:**

Options:

Netflix Eureka Service Discovery,

etcd,

Consul,

Apache Zookeeper.

Also, some systems do not have an explicit service registry. Instead these use an interservice communication infrastructure known as a service mesh.

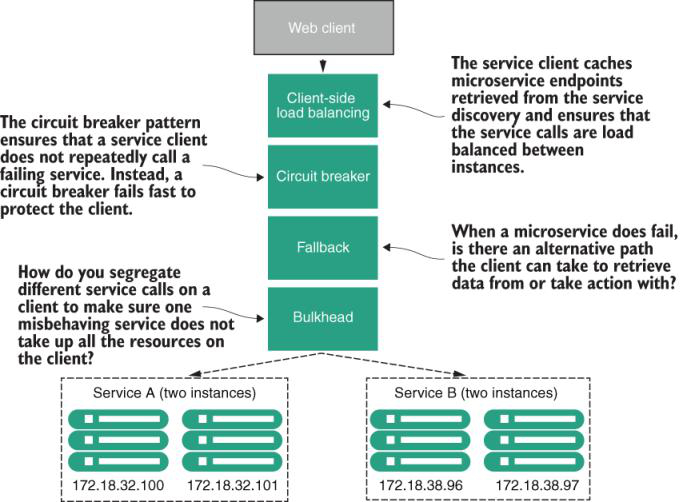
**Service Routing:**

Spring Cloud API Gateway

**What is microservice resiliency pattern?**

Because microservice architectures are highly distributed, you have to be extremely sensitive in how you prevent a problem in a single service (or service instance) from cascading up and out to the consumers of the service.

With microservices, you must protect the service caller from a poorly behaving service. Remember, a slow or down service can cause disruptions beyond the immediate service.



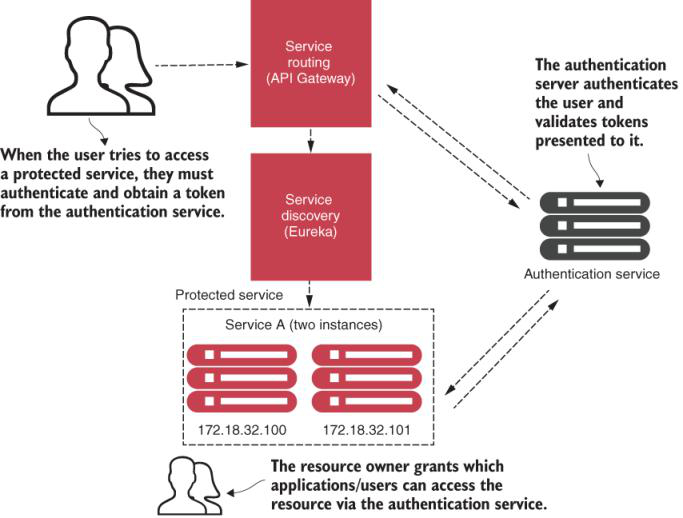
**What is Oauth2?**

OAuth2 is a token-based security framework that allows a user to authenticate themselves with a third-party authentication service. If the user successfully authenticates, they will be presented with a token that must be sent with every request.

The main goal behind OAuth2 is that when multiple services are called to fulfill a user’s request, the user can be authenticated by each service without having to present their credentials to each service processing their request.

**What is Microservice Service Security pattern?**

* Authentication—How you determine the service client calling the service is who they say they are.
* Authorization—How you determine whether the service client calling a microservice is allowed to undertake the action they’re trying to take.
* Credential management and propagation—How you prevent a service client from constantly having to present their credentials for service calls involved in a transaction. To achieve this, we’ll look at how you can use token-based security standards such as OAuth2 and JSON Web Tokens (JWT) to obtain a token that can be passed from service call to service call to authenticate and authorize the user.
* Using a token-based security scheme, you can implement service authentication and authorization without passing around client credentials.



**What is Microservice Logging and Tracing Patterns?**

The downside of a microservice architecture is that it’s much more difficult to debug, trace, and monitor the issues because one simple action can trigger numerous microservice calls within your application.

We achieve distributed tracing with Spring Cloud Sleuth, Zipkin, and the ELK Stack

we’ll look at the following three core logging and tracing patterns to achieve distributed tracing:

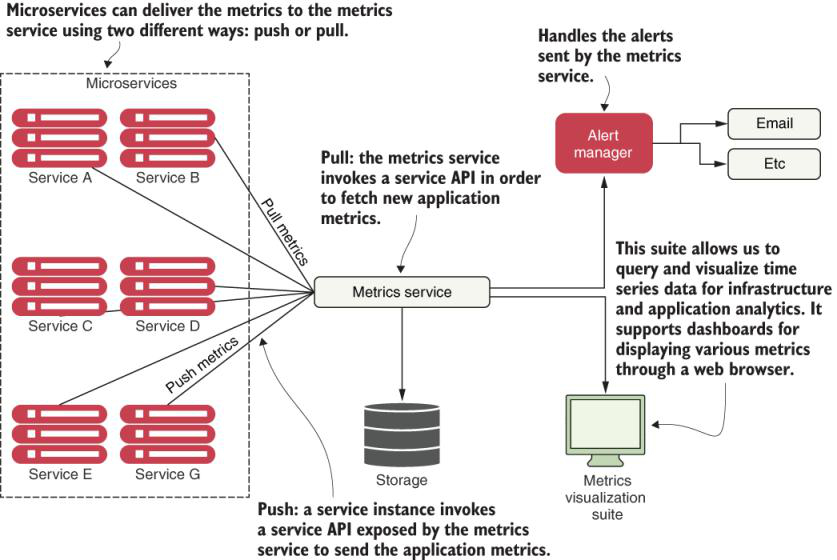
* Log correlation—How you tie together all the logs produced between services for a single user transaction. With this pattern, we’ll look at how to implement a correlation ID, which is a unique identifier that’s carried across all service calls in a transaction and that can be used to tie together log entries produced from each service.
* Log aggregation—With this pattern, we’ll look at how to pull together all of the logs produced by your microservices (and their individual instances) into a single queryable database across all the services involved and understand the performance characteristics of the services in the transaction.
* Microservice tracing—We’ll explore how to visualize the flow of a client transaction across all the services involved and understand the performance characteristics of the transaction’s services.
* A well-thought-out logging and tracing strategy makes debugging transactions across multiple services manageable.

A diagram of a logistic process

Description automatically generated

**What is application metric pattern?**

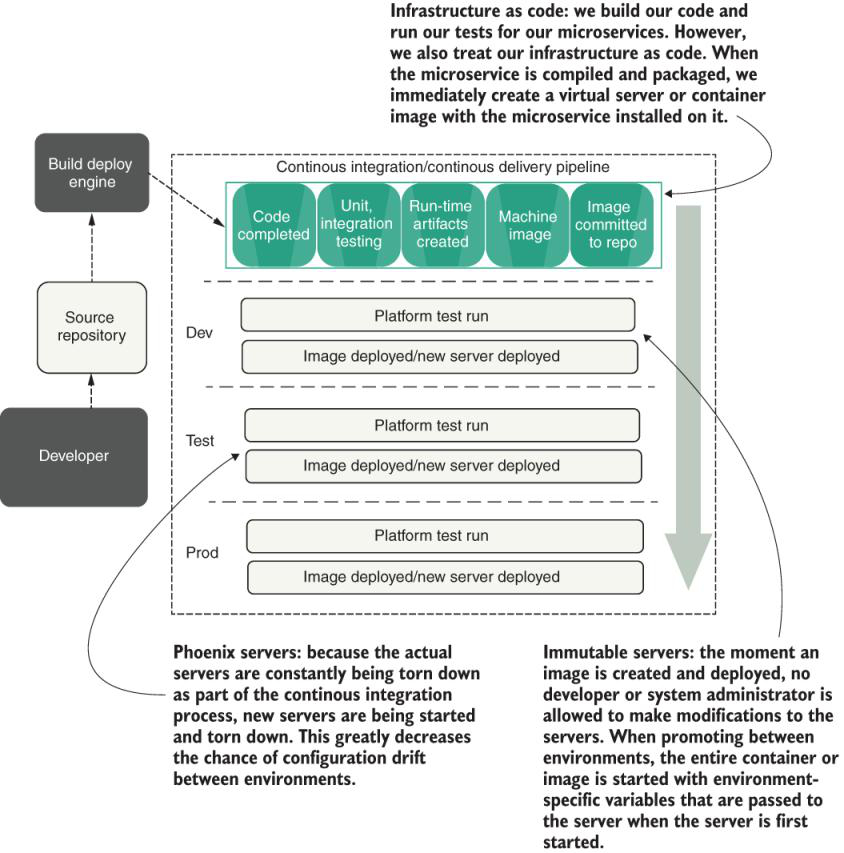
Metrics are pulled or pushed from the microservices and collected and stored in the metrics service to be shown using a metrics visualization suite and an alert management tool



**What is Microservice Build/Deployment Patterns?**

You want the deployment of the microservice and the server it’s running on to be one atomic artifact that’s deployed as a whole between environments.

Our goal with these patterns and topics is to ruthlessly expose and stamp out configuration drift as quickly as possible before it can hit your upper environments.



**What is one downside of being distributed?**

The more distributed a system is, the more places it can fail.

**Why Spring cloud?**

With a microservice architecture, we have more points of failure. That’s because instead of having a single monolith application, we now have an ecosystem of multiple individual services that interact with each other. This is the main reason why developers often encounter different administration and synchronization challenges or points of failure when creating microservice applications or architectures. To avoid possible points of failure, we will use Spring Cloud. Spring Cloud offers a set of features (service registration and discovery, circuit breakers, monitoring, and others) that will allow us to quickly build microservice architectures with minimal configurations.

**What is Spring Cloud?**

With Spring Cloud, we can map the technologies we’re going to use directly to the microservice patterns we’ve explored so far.



**Spring Cloud Config**

Integrates well with:

Git

An open source version control system that lets you manage and track changes to any text file. Spring Cloud Config integrates with a Git backend repository and reads the application’s configuration data from the repository.

Consul

An open source service discovery that allows service instances to register themselves with a service. Service clients can then query Consul to find the location of their service instances. Consul also includes a key-value store database that Spring Cloud Config uses to store application configuration data.

Eureka

An open source Netflix project that, like Consul, offers similar service discovery capabilities. Eureka also has a key-value database that can be used with Spring Cloud Config.

**Spring Cloud Service Discovery**

Consul

Zookeeper

Eureka

Notes:

Although Consul and Zookeeper are powerful and flexible, the Java developer community still uses Eureka. This book contains examples with Eureka to keep it manageable and material focused, and to reach the largest audience possible

**Spring Cloud API Gateway**

The API Gateway provides service-routing capabilities for your microservice application. Like the name says, it is a service gateway that proxies service requests and makes sure that all calls to your microservices go through a single “front door” before the targeted service is invoked. With this centralization of service calls, you can enforce standard service policies such as security authorization, authentication, content filtering, and routing rules. You can implement the API Gateway using Spring Cloud Gateway (<https://spring.io/projects/spring-cloud-gateway>).

#### Note

#### In this book, we use the Spring Cloud API Gateway that was built with Spring Framework 5 Project Reactor (allowing integration with Spring Web Flux) and Spring Boot 2 to better integrate our Spring projects.

**Spring Cloud Stream**

Spring Cloud Stream (https://cloud.spring.io/spring-cloud-stream) is an enabling technology that lets you easily integrate lightweight message processing into your microservice. Using Spring Cloud Stream, you can build intelligent microservices that use asynchronous events as these occur in your application. You can also quickly integrate your microservices with message brokers such as RabbitMQ (https://www.rabbitmq.com) and Kafka.

**Spring Cloud Sleuth**

Spring Cloud Sleuth (https://cloud.spring.io/spring-cloud-sleuth/) **lets you integrate unique tracking identifiers into the HTTP calls and message channels (RabbitMQ, Apache Kafka) used within your application**. These tracking numbers, sometimes referred to as correlation or trace IDs, allow you to track a transaction as it flows across the different services in your application. With Spring Cloud Sleuth, trace IDs are automatically added to any logging statements you make in your microservice.

**The real beauty of Spring Cloud Sleuth is seen when it’s combined with logging-aggregation technology tools like the ELK Stack (https://www.elastic.co/what-is/elk-stack) and tracking tools like Zipkin** (http://zipkin.io). Open Zipkin takes data produced by Spring Cloud Sleuth and allows you to visualize the flow of your service calls involved for a single transaction. The ELK Stack is the acronym for three open source projects:

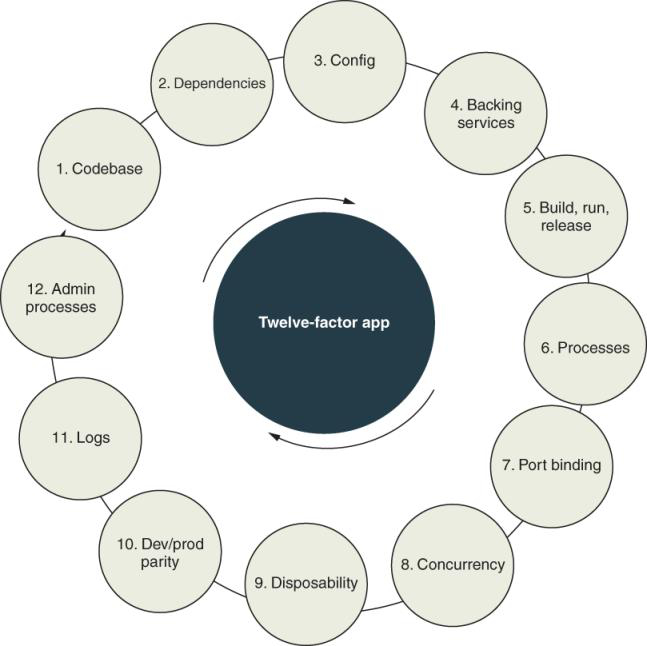
Elasticsearch (https://www.elastic.co) is a search and analytics engine.

Logstash (https://www.elastic.co/products/logstash) is a server-side, data-processing pipeline that consumes data and then transforms it in order to send it to a “stash.”

Kibana (https://www.elastic.co/products/kibana) is a client UI that allows the user to query and visualize the data of the whole stack.

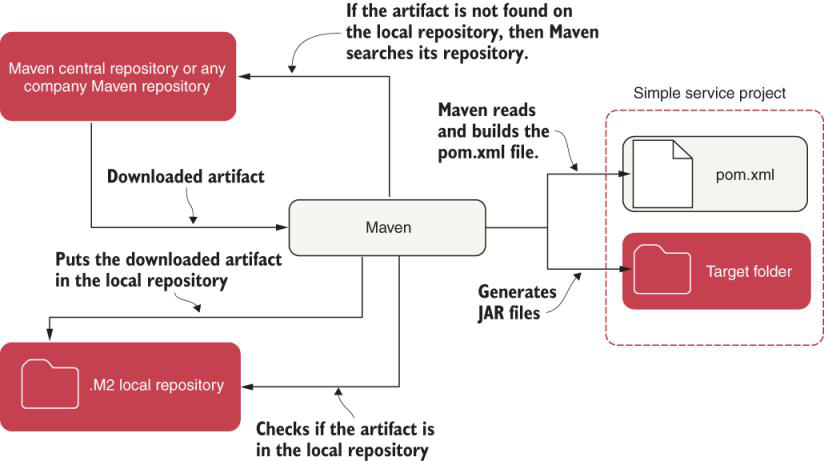
**What are 12-factor application best practices?**

Twelve-factor application best practices



**How Maven build tool works?**

Maven reads the dependencies stored in the pom.xml file and then searches for them in a local repository. If the dependencies are not found, then Maven downloads the dependencies from the Maven repository, inserting them into your local repository



**What is RepresentationModel class from Spring Hateos project?**

import org.springframework.hateoas.RepresentationModel;

RepresentationModel<License> gives us the ability to add links to the License model class. Now that we have everything set up, let’s create the HATEOS configuration to retrieve the links for the LicenseController class. The next listing shows how this is done. For this example, we’re only going to change the getLicense() method in the LicenseController class.

#### Listing 3.11: Adding links to the LicenseController

@RequestMapping(value="/{licenseId}",method = RequestMethod.GET)

public ResponseEntity<License> getLicense(

@PathVariable("organizationId") String organizationId,

@PathVariable("licenseId") String licenseId) {

License license = licenseService.getLicense(licenseId,

organizationId);

license.add(linkTo(methodOn(LicenseController.class)

.getLicense(organizationId, license.getLicenseId()))

.withSelfRel(),

linkTo(methodOn(LicenseController.class)

.createLicense(organizationId, license, null))

.withRel("createLicense"),

linkTo(methodOn(LicenseController.class)

.updateLicense(organizationId, license))

.withRel("updateLicense"),

linkTo(methodOn(LicenseController.class)

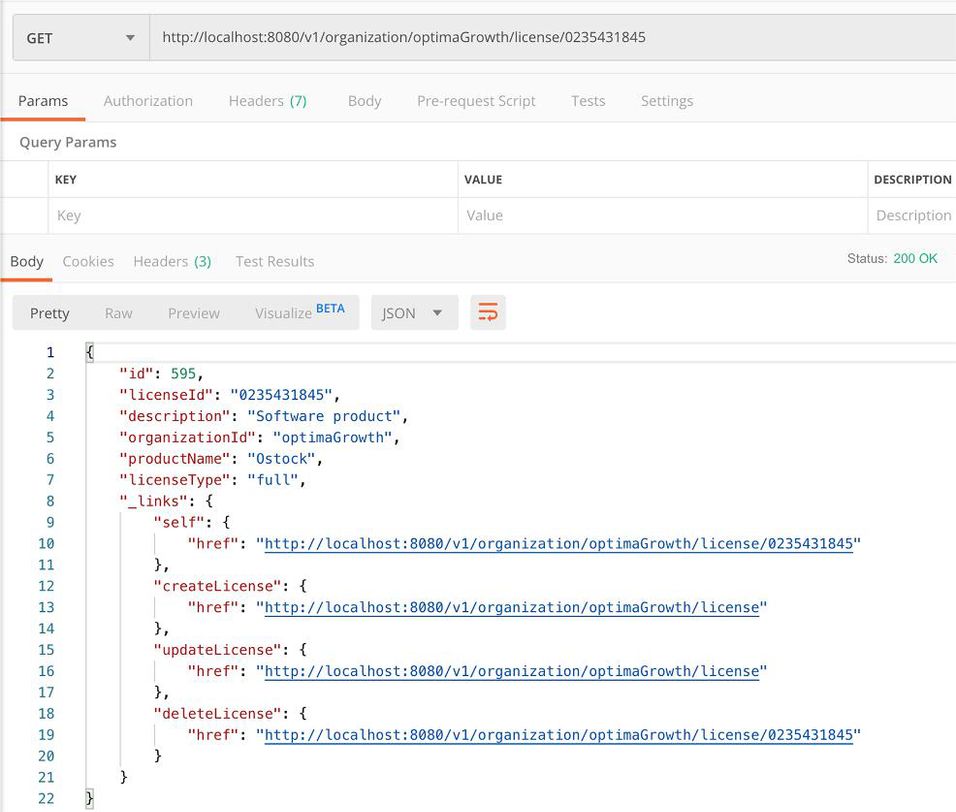
.deleteLicense(organizationId, license.getLicenseId()))

.withRel("deleteLicense"));

return ResponseEntity.ok(license);

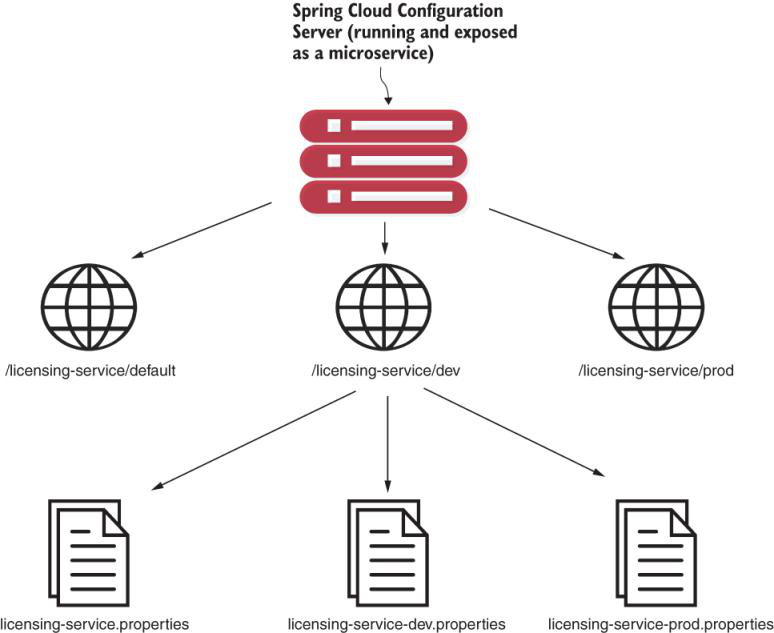
}

HATEOAS links on the response body of the HTTP GET license service.



**How Spring cloud config work?**

Spring Cloud Config exposes environment-specific properties as HTTP-based endpoints.



Spring Cloud Config exposes environment-specific properties as HTTP-based endpoints



Retrieving configuration information for the licensing service using the dev profile

A screenshot of a computer program

Description automatically generated

**How environment related configuration are resolved in Spring cloud?**

If we look closely, we will see that when we selected the dev endpoint, the Spring Cloud Configuration Server returned both the default configuration properties and the dev licensing service configuration. The reason why Spring Cloud Config returns both sets of configuration information is that the Spring framework implements a hierarchical mechanism for resolving problems. When the Spring framework does this, it looks for the property defined in the default properties file first and then overrides the default with an environment-specific value if one is present. In concrete terms, if you define a property in the licensing-service.properties file and don’t define it in any of the other environment configurations (for example, licensing-service-dev.properties), the Spring framework will use the default value.

**How YAML correlated to properties file?**

The Spring Boot application supports two mechanisms to define a property: YAML (YAML Ain’t Markup Language) and a dot-separated property name. We will choose YAML as the means for configuring our application. The hierarchical format of YAML property values maps directly to these names: spring.application.name, spring.profiles.active, and spring .cloud.config.uri.

**How to start a Spring microservice by providing JVM arguments?**

mvn spring-boot:run

By running this command without any properties set, the licensing server automatically attempts to connect to the Spring Cloud Configuration Server using the endpoint (in this case, our endpoint is http://localhost:8071) and the active profile (dev) defined previously in the bootstrap.yml file of the licensing service.

If you want to override these default values and point to another environment, you can do so by compiling the licensing service project down to a JAR file, and then run the JAR with a D system property override. The following command-line call demonstrates how to launch the licensing service, passing all the commands via JVM arguments:

java -Dspring.cloud.config.uri=http://localhost:8071 \

-Dspring.profiles.active=dev \

-jar target/licensing-service-0.0.1-SNAPSHOT.jar

This example demonstrates how to override Spring properties via the command line.

With this command line, we override these two parameters:

spring.cloud.config.uri

spring.profiles.active

**How can we configure Spring actual to show only relevant information and not sensitive information?**

Spring Boot provides a wealth of capabilities on how to configure what information is returned by the Spring Actuator endpoints. That’s outside the scope of this book, however. Craig Walls’ excellent book, Spring Boot in Action (Manning, 2016), covers this subject in detail. We highly recommend that you review your corporate security policies and Walls’ book to provide the right level of information you want to expose through Spring Actuator.

**How do you Refresh Your Properties Using Spring Cloud Config Server?**

One of the first questions that comes up from development teams when they want to use the Spring Cloud Configuration Server is how can they dynamically refresh their applications when a property changes. Rest assured. The Config Server always serves the latest version of a property. Changes made to a property via its underlying repository will be up to date!

Spring Boot applications, however, only read their properties at startup, so property changes made in the Config Server won’t be automatically picked up by the Spring Boot application. But Spring Boot Actuator offers a @RefreshScope annotation that allows a development team to access a /refresh endpoint that will force the Spring Boot application to reread its application configuration. The following listing shows this annotation in action.

package com.optimagrowth.license;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.context.config.annotation.RefreshScope;

@SpringBootApplication

@RefreshScope

public class LicenseServiceApplication {

public static void main(String[] args) {

SpringApplication.run(LicenseServiceApplication.class, args);

}

}

Note a couple of things about the @RefreshScope annotation. This annotation only reloads the custom Spring properties you have in your application configuration. Items like your database configuration used by Spring Data won’t be reloaded by this annotation.

**What are different approaches to refresh configurations from Spring cloud config server?**

1. Using Spring cloud bus

Spring Cloud Bus requires an extra piece of running middleware: RabbitMQ. This is an extremely useful means of detecting changes, but not all Spring Cloud Config backends support the push mechanism.

2. One technique that we’ve used to handle application configuration refresh events is to refresh the application properties in Spring Cloud Config. Then we write a simple script to query the service discovery engine to find all instances of a service and call the /refresh endpoint directly.

You can also restart all the servers or containers to pick up the new property. This is a trivial exercise, especially if you’re running your services in a container service such as Docker. Restarting Docker containers literally takes seconds and will force a reread of the application configuration.

**What are different ways of storing configuration data for Spring cloud config server?**

* 1. Using file system
  2. On a Git repo
  3. On  HashiCorp Vault for defining secrets as any piece of information we want to restrict or control access to, such as passwords, certificates, API keys, and so forth.

**How do encrypt/decrypt configuration properties?**

#### Listing 5.15: Setting a symmetric key in the boostrap.yml file

cloud:

config:

server:

native:

search-locations: classpath:/config

git:

uri: https://github.com/ihuaylupo/config.git

searchPaths: licensingservice

server:

port: 8071

encrypt:

key: secretkey ❶

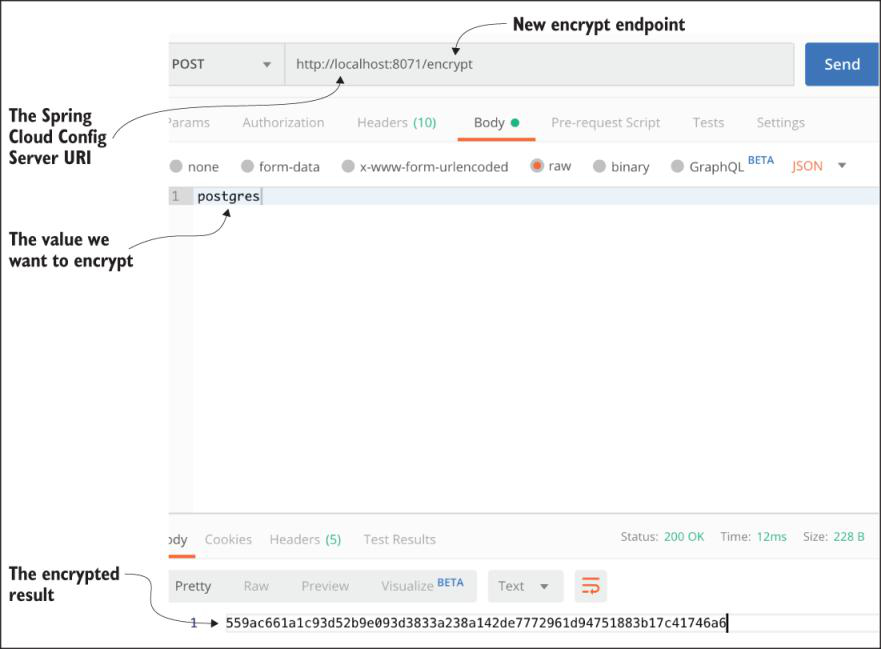
❶ Tells the Config Server to use this value as the symmetric key

When you fire up your Spring Cloud Config instance, Spring Cloud Config detects that the ENCRYPT\_KEY environment variable or the bootstrap file property is set, and automatically adds two new endpoints, /encrypt and /decrypt to the Spring Cloud Config service. We will use the /encrypt endpoint to encrypt the postgres value. [Figure 5.13](https://cdn2.percipio.com/1694791404.991d230efcabde6670c77cd268ad2ef7989585b4/eod/books/157553/OEBPS/section-54-47.xhtml#ch5fig13) shows how to encrypt the postgres value using the /encrypt endpoint and Postman.

#### Note

When you call the /encrypt or /decrypt endpoints, you need to make sure you do a POST to these endpoints.

ncrypting the Spring data source password using the /encrypt endpoint



#### Adding an encrypted value to the licensing service properties file

spring.datasource.url = jdbc:postgresql://localhost:5432/ostock\_dev

spring.datasource.username = postgres

spring.datasource.password = {cipher}

➥ 559ac661a1c93d52b9e093d3833a238a142de7772961d94751883b17c41746a6