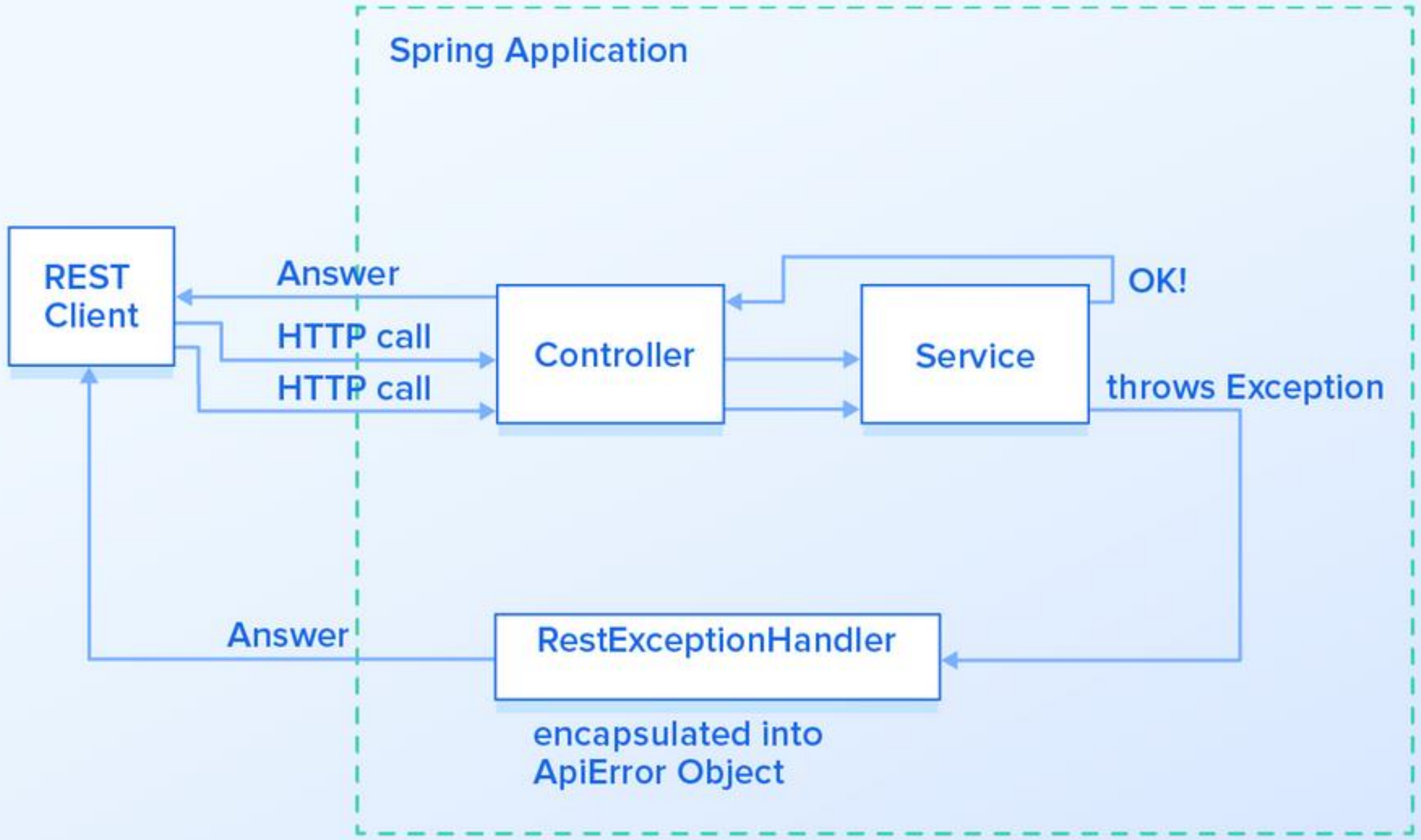
Session Content (16)

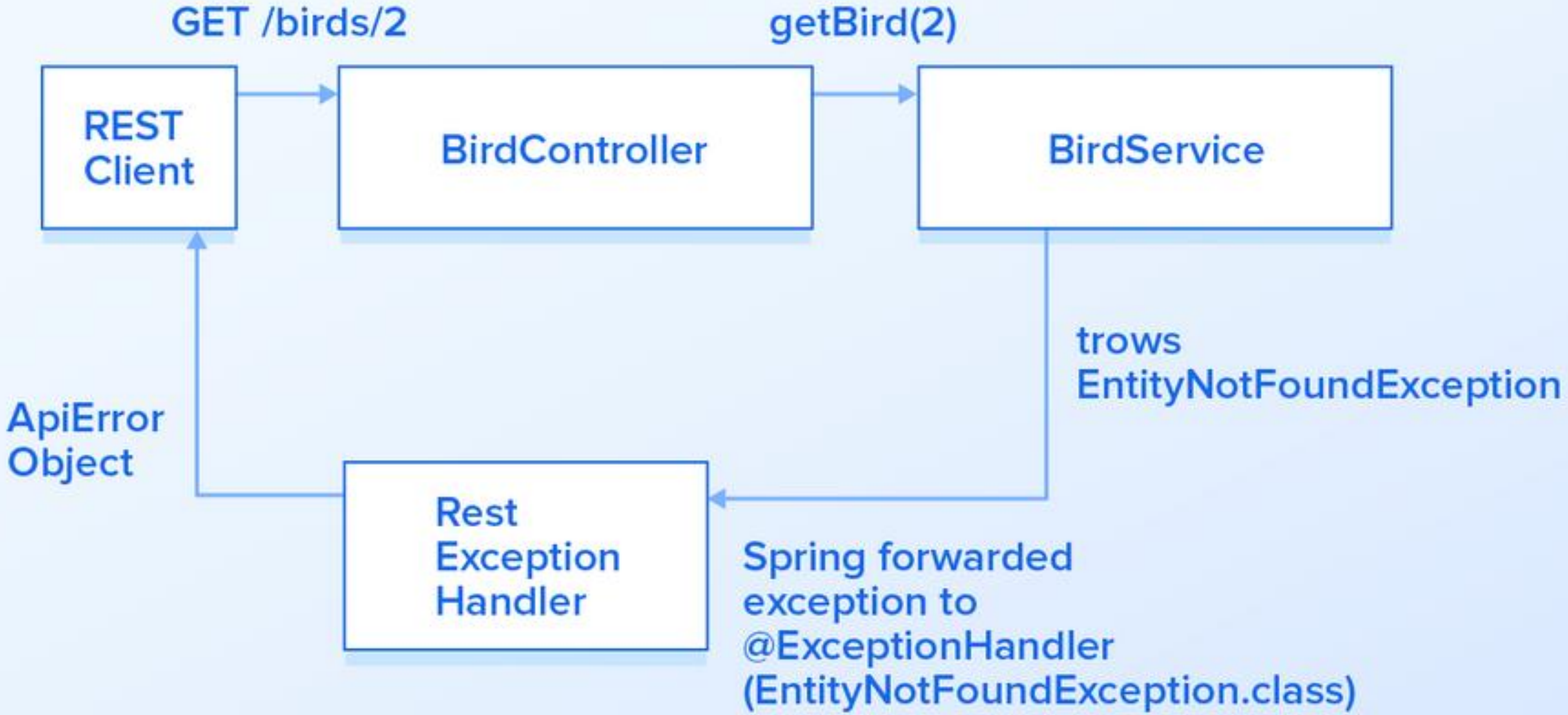
- Spring MongoDB Practices
- Spring boot Exception Handling
- Exception Handling Practices
- Spring Boot Internationalization

Spring Boot - Exception Handling

- Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

- Controller Advice:- The @ControllerAdvice is an annotation, to handle the exceptions globally.
- Exception Handler:- The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.


Spring Application
REST Client
Answer
HTTP call
HTTP call
Controller
Service
OK!
throws Exception
Answer
RestExceptionHandler
encapsulated into ApiError Object


GET /birds/2
getBird(2)
REST Client
BirdController
BirdService
trows EntityNotFoundException
ApiError Object
Rest Exception Handler
Spring forwarded exception to @ExceptionHandler (EntityNotFoundException.class)

Session Content (17)

- Spring Boot Internationalization
- Logging in spring boot
- Custom Logging with Logback
- Logging with Log4j2

Spring Boot - Internationalization

- Internationalization is a process that makes your application adaptable to different languages and regions without engineering changes on the source code. In ether words, Internationalization is a readiness of Localization.
- The file for the default locale will have the name *messages.properties*, and files for each locale will be named *messages_XX.properties*, where *XX* is the locale code.
- The keys for the values that will be localized have to be the same in every file, with values appropriate to the language they correspond to.
- If a key does not exist in a certain requested locale, then the application will fall back to the default locale value.

Logging in spring boot

- **Logging in spring boot:-** is very flexible and easy to configure. Spring boot supports various logging providers through simple configurations.
- **Default Zero Configuration Logging:-** Spring boot's active enabled logging is determined by spring-boot-starter-logging artifact and its auto-configuration that enables any one of the supported logging providers (Java UtilLogging, Log4J2, and Logback) based on the configuration provided.
- **Default Logging Provider is Logback:-** If we do not provide any logging-specific configuration, we will still see logs printed in "console" because default logging uses Logback to log DEBUG messages into the Console.
- Spring boot's internal logging is written with Apache Commons Logging so it is one and only mandatory dependency. Till Spring boot version 1.x – we had to import commons-logging manually. Since boot 2.x, it is downloaded transitively.
- **Log Statements using SLF4J:-** To add log statements in application code, use *org.slf4j.Logger* and *org.slf4j.LoggerFactory* from SLF4J. It provides lots of useful methods for logging and also decouples the logging implementation from the application.

Custom Logging with Logback

- The default logging is good enough for getting started and POC purposes. But in real-life enterprise applications, we need more fine control over logging with other complex requirements. In that case, having a dedicated logging configuration is suitable.
- Spring boot by default uses logback, so to customize its behavior, all we need to **add logback.xml in classpath** and define customization over the file

Logging with Log4j2

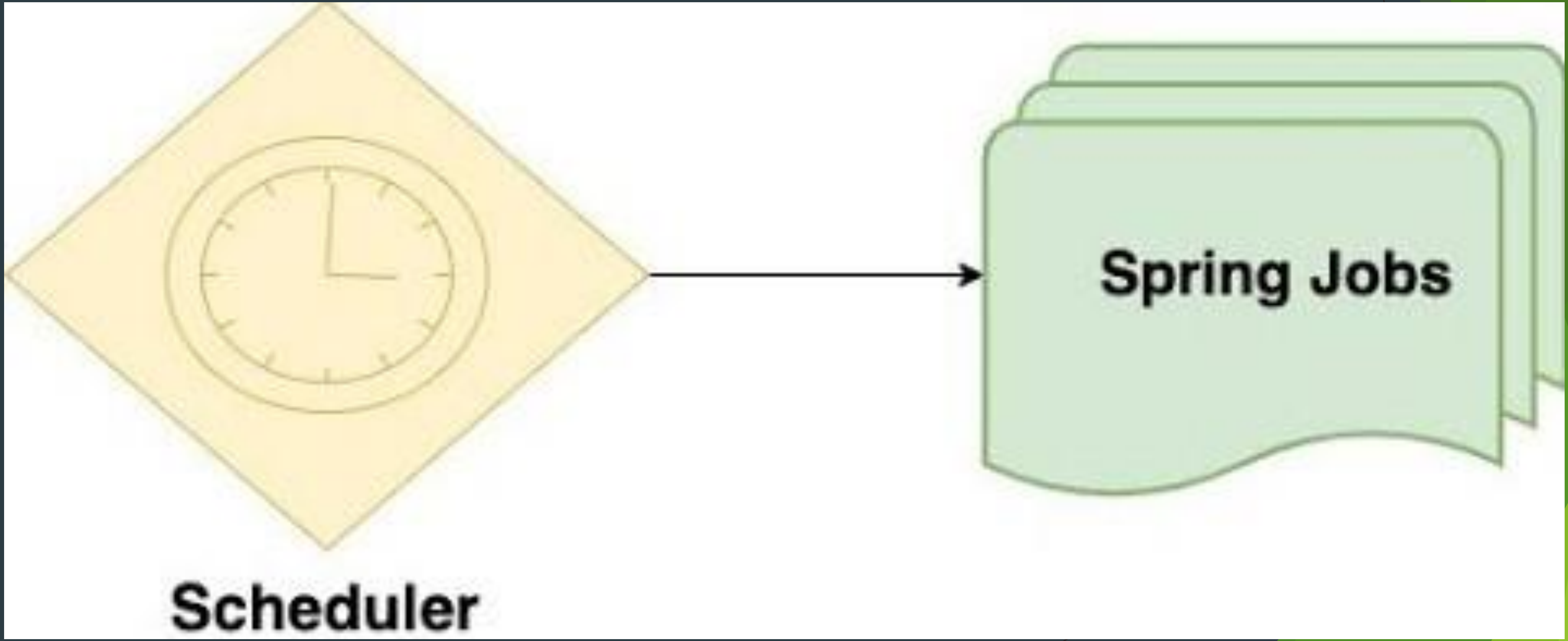
- **Excluding Logback and Including Log4j2:-** To exclude default logging, exclude spring-boot-starter-logging dependency and explicitly import add spring-boot-starter-log4j2 to the classpath.
- **Add Log4j2 Configuration File:-** add log4j2 specific configuration file in the classpath (typically in resources folder). It can be named as any of the following:
  - log4j2-spring.xml
  - log4j2.xml

Session Content (18)

- **Spring Boot Scheduled Jobs**
- **@Async annotation**
- Spring Boot – DB Migration (Flyway)

Scheduled Jobs in Spring Boot

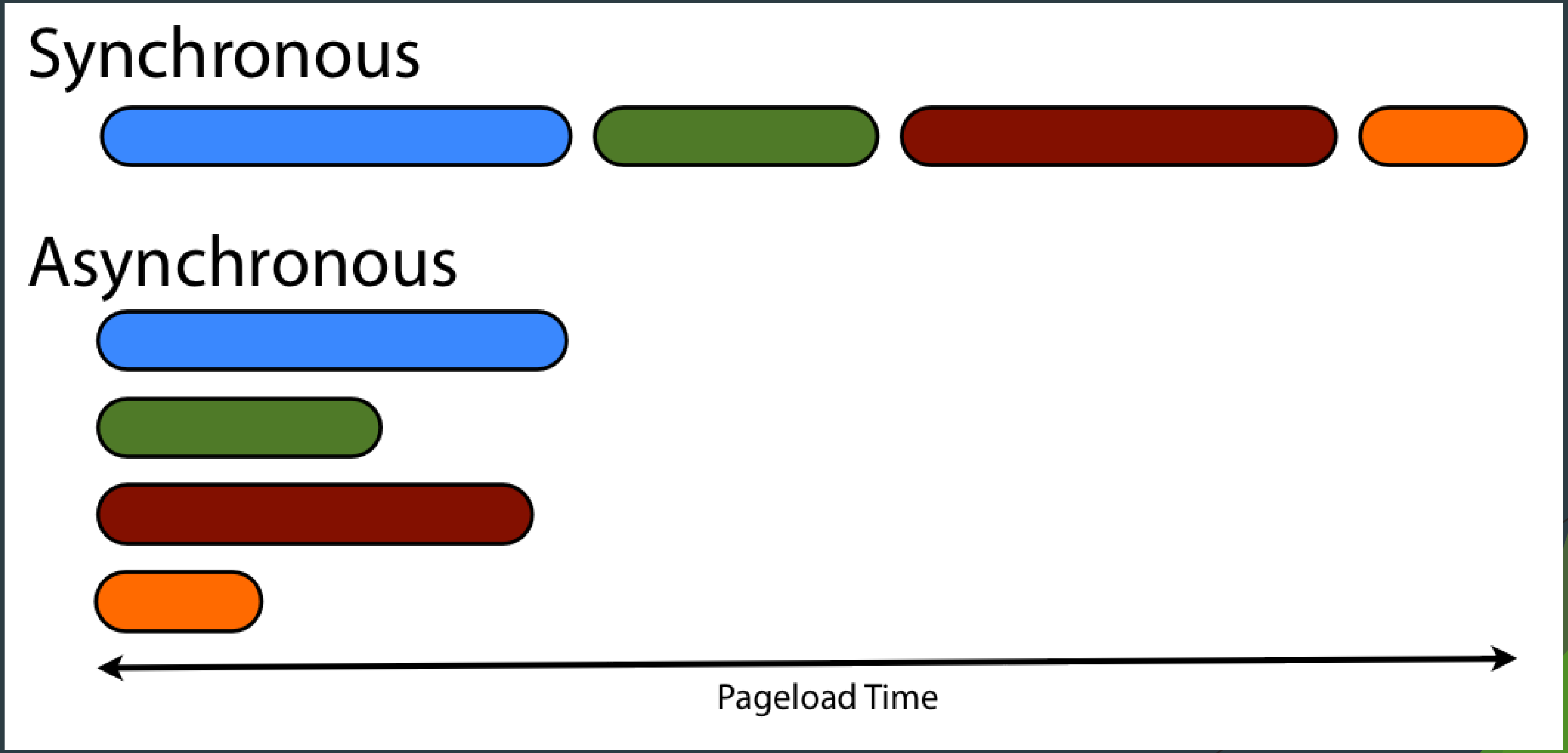
- Scheduling is the process of executing a piece of logic at a specific time in the future. Scheduled jobs are a piece of business logic that should run on a timer. Spring allows us to run scheduled jobs in the Spring container by using some simple annotations.



- Scheduling is part of the core module, so we do not need to add any dependencies.
- Scheduling is not enabled by default. We explicitly enable scheduling by adding the @EnableScheduling annotation to a Spring configuration class.
- We can make the scheduling conditional on a property so that we can enable and disable scheduling by setting the property.
- We create scheduled jobs by decorating a method with the @Scheduled annotation.
- Only methods with void return type and zero parameters can be converted into scheduled jobs by adding @Scheduled annotation.
- We set the interval of executing by specifying the fixedRate or fixedDelay attribute in the @Scheduled annotation.
- We can choose to delay the first execution of the method by specifying the interval using the initialDelay attribute.
- We can deploy multiple Scheduler Instances using the ShedLock library which ensures only one instance to run at a time by using a locking mechanism in a shared database.
- We can use a Distributed Job Scheduler like Quartz to address more complex scenarios of scheduling like resuming failed jobs, and reporting.

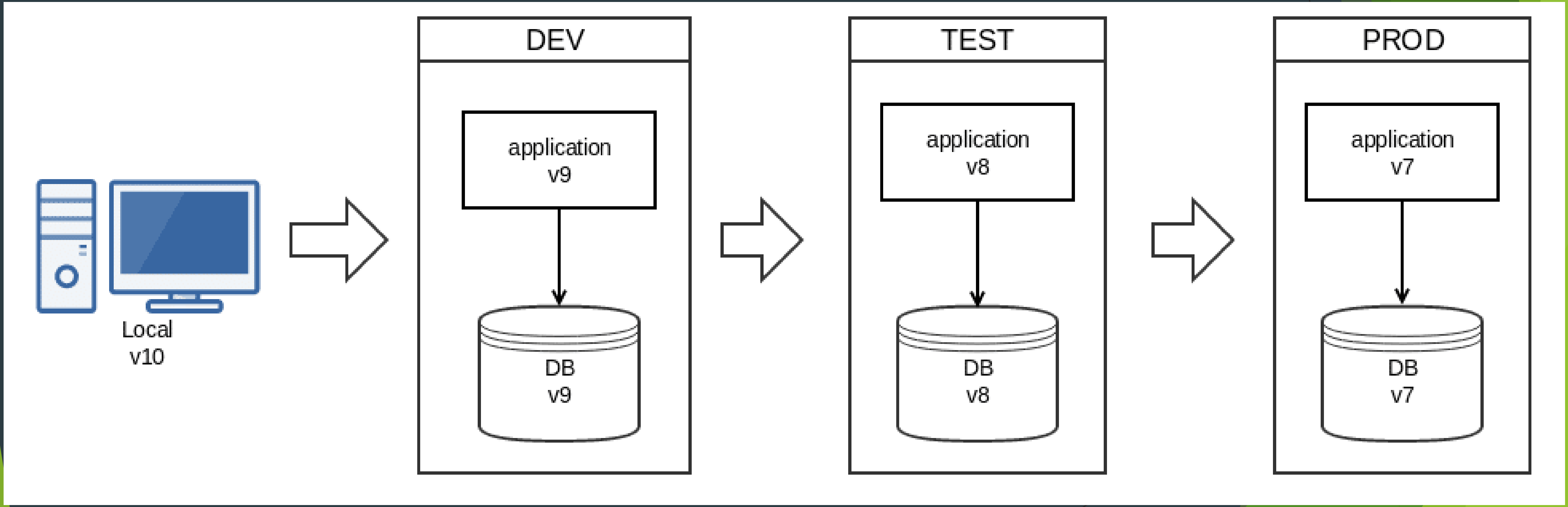
@Async annotation

- *Spring provides a feature to run a long-running process in a separate thread.* This feature is helpful when scaling services. By using the *@Async and @EnableAsync annotations*, we can run the run expensive jobs in the background and wait for the results by using Java’s CompletableFuture interface.



Spring Boot - Flyway

- Spring Boot simplifies database migrations by providing integration with Flyway, one of the most widely used database migration tools.
- Flyway is a version control application to evolve your Database schema easily and reliably across all your instances.



Why Do We Need Database Migrations?

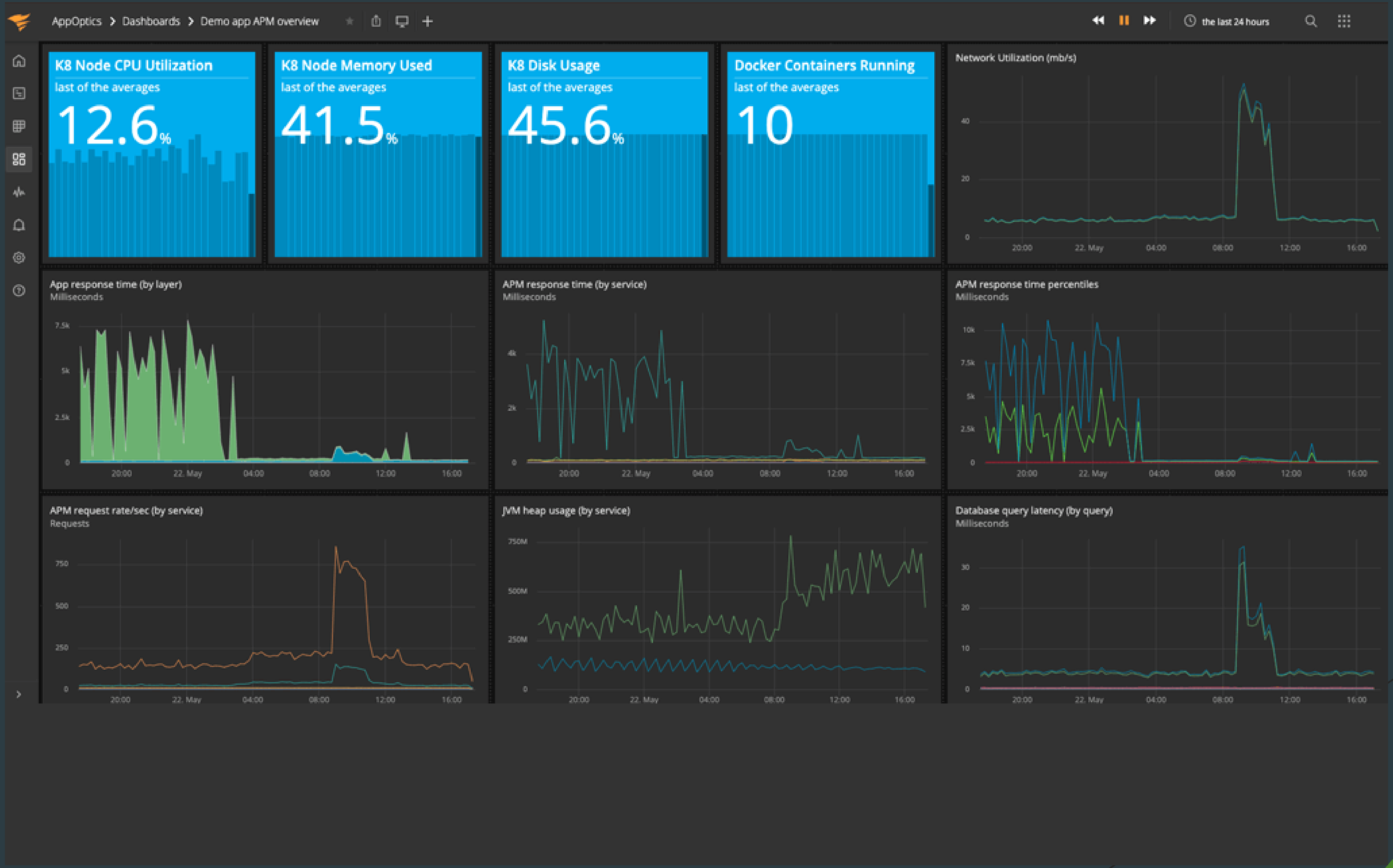
- I’ve worked on a project where all database changes were deployed manually. Over time, more people joined and, naturally, they started asking questions:
  - What state is the database in on this environment?
  - Has a specific script already been applied or not?
  - Has this hotfix in production been deployed in other environments afterward?
  - How can I set up a new database instance to a specific or the latest state?
- Answering these questions required one of us to check the SQL scripts to find out if someone has added a column, modified a stored procedure, or similar things. If we multiply the time spent on all these checks with the number of environments and add the time spent on aligning the database state, then we get a decent amount of time lost.
- Automatic database migrations with Flyway or similar tools allow us to:
  - Create a database from scratch.
  - Have a single source of truth for the version of the database state.
  - Have a reproducible state of the database in local and remote environments.
  - Automate database changes deployment, which helps to minimize human errors.

How Flyway Works

- To keep track of which migrations we've already applied and when, it adds a special bookkeeping table to our schema. This metadata table also tracks migration checksums, and whether or not the migrations were successful.
- The framework performs the following steps to accommodate evolving database schemas:
  - It checks a database schema to locate its metadata table (*SCHEMA_VERSION* by default). If the metadata table doesn't exist, it will create one.
  - It scans an application classpath for available migrations.
  - It compares migrations against the metadata table. If a version number is lower or equal to a version marked as current, it's ignored.
  - It marks any remaining migrations as pending migrations. These are sorted based on the version number and are executed in order.
  - As each migration is applied, the metadata table is updated accordingly.

Spring Boot - Actuator

- **Spring Boot Actuator** is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live).



AppOptics > Dashboards > Demo app APM overview
the last 24 hours
K8 Node CPU Utilization
last of the averages
12.6%
K8 Node Memory Used
last of the averages
41.5%
K8 Disk Usage
last of the averages
45.6%
Docker Containers Running
last of the averages
10
Network Utilization (mb/s)
App response time (by layer)
Milliseconds
APM response time (by service)
Milliseconds
APM response time percentiles
Milliseconds
APM request rate/sec (by service)
Requests
JVM heap usage (by service)
Database query latency (by query)
Milliseconds

Spring Boot Actuator Features

- **Endpoints** Actuator endpoints allow you to monitor and interact with your application. Spring Boot includes a number of built-in endpoints and you can also add your own. For example the health endpoint provides basic application health information. Run up a basic application and look at /actuator/health.

- **Metrics** Spring Boot Actuator provides dimensional metrics by integrating with Micrometer.

- **Audit** Spring Boot Actuator has a flexible audit framework that will publish events to an AuditEventRepository. Once Spring Security is in play it automatically publishes authentication events by default. This can be very useful for reporting, and also to implement a lock-out policy based on authentication failures.

Session Content (20)

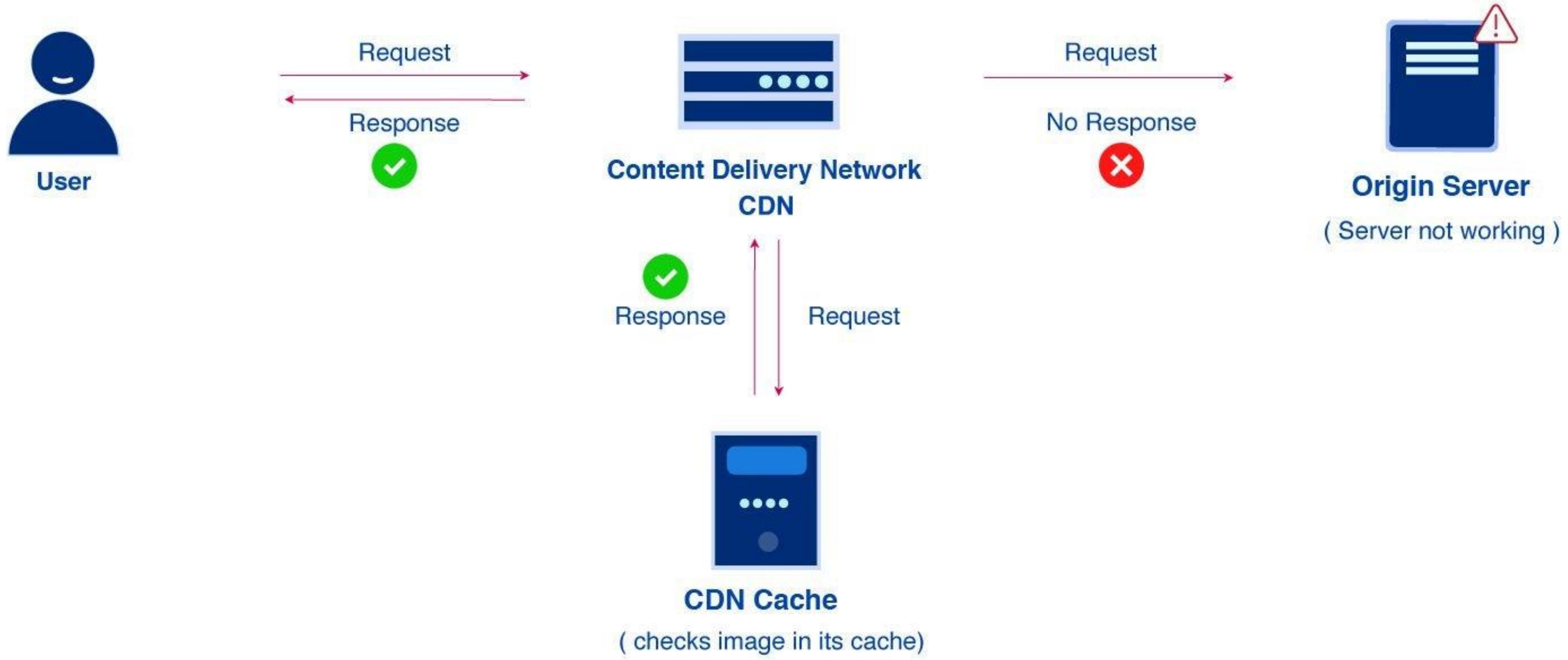
- Content Delivery Network (CDN)
- Store Files on server
- Overview for AWS Storage
- Overview for Google Cloud Storage
- RESTful API Documentation (Swagger UI)

Content Delivery Network (CDN)

- A content delivery network (CDN) refers to a geographically distributed group of servers which work together to provide fast delivery of Internet content.

- A CDN allows for the quick transfer of assets needed for loading Internet content including HTML pages, javascript files, stylesheets, images, and videos.

- Benefits of using a CDN:-
  - **Improving website load times**
  - **Reducing bandwidth costs**
  - **Increasing content availability and redundancy**


User
Request
Response
Content Delivery Network
CDN
Request
No Response
Origin Server
(Server not working)
Response
Request
CDN Cache
(checks image in its cache)

Session Content (21)

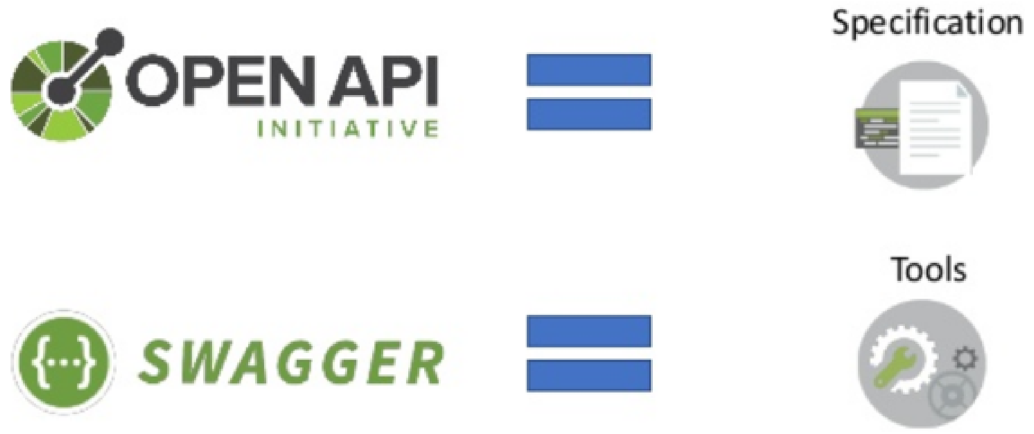
- RESTful API Documentation (Swagger UI)
- Consuming REST endpoints with RestTemplate

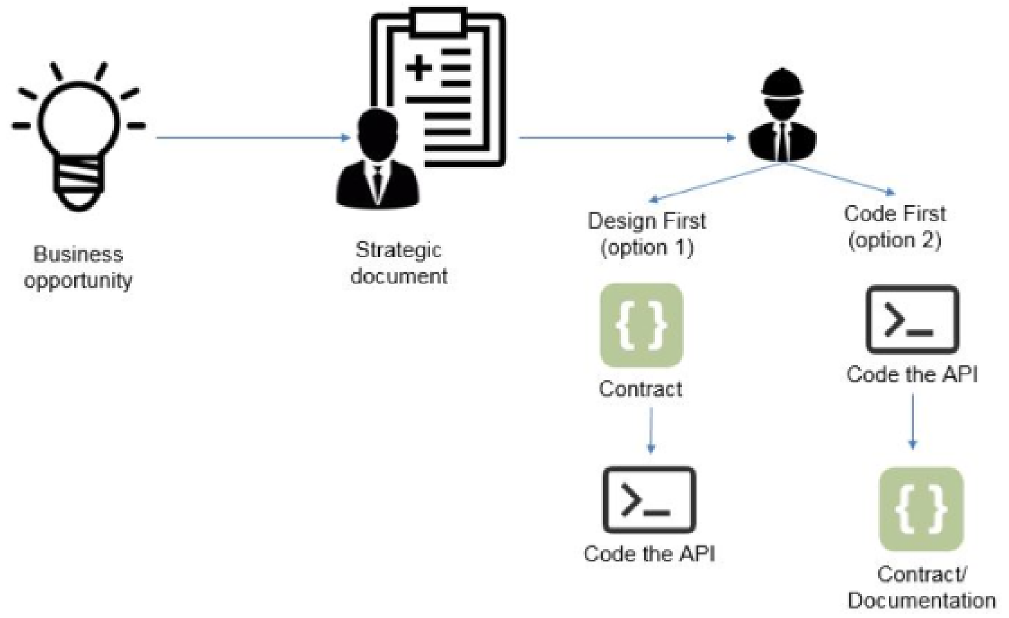
RESTful API Documentation (Swagger UI)

- **Swagger** is an Interface Description Language for describing RESTful APIs expressed using JSON.
- **Swagger** is used together with a set of open-source software tools to design, build, document, and use RESTful web services.
- **Swagger** includes automated documentation, code generation (into many programming languages), and test-case generation.
- Swagger's open-source tooling usage can be broken up into different use cases: development, interaction with APIs, and documentation.
- When creating APIs, Swagger tooling may be used to automatically generate an Open API document based on the code itself. This embeds the API description in the source code of a project and is informally called code-first or bottom-up API development.
- Swagger UI allows anyone — be it your development team or your end consumers — to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI (formerly known as Swagger) Specification, with the visual documentation making it easy for back end implementation and client side consumption.

What Is OpenAPI 3?

- OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs.
- An OpenAPI file allows you to describe:
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters Input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
- API specifications can be written in YAML or JSON.





Why springdoc-openapi?

- Many industry standards are using OpenAPI format: Open Banking, PSD2, FHIR, ISO 20022, BIAN, IATA, and others.
- All the modern API Management solutions are supporting OpenAPI 3 format, and the vendors prefer this format to benefit from all the functionalities of their solution.
- The existing Open Source libraries that supported both swagger and spring-boot, has not evolved with new standard OpenAPI 3 which has been released since July 2017.
- springdoc-openapi java library helps to automate the generation of API documentation using spring boot projects.
- This library supports:
  - OpenAPI 3
  - Spring-boot (v1 and v2)
  - JSR-303, specifically for @NotNull, @Min, @Max, and @Size.
  - Swagger-ui
  - OAuth 2

Integration with spring-web

- For spring-web projects, they just need to add the springdoc-openapi dependency
- All the spring-mvc annotations are by default supported: @RequestParam, @PathVariable, @RequestBody, …
- And also ignores default spring-mvc types (HttpServletRequest, HttpServletResponse, …)

```xml
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>${springdoc.version}</version>
</dependency>
```

Consuming REST endpoints with RestTemplate

- There’s a lot that goes into interacting with a REST resource from the client’s perspective mostly tedium and boilerplate. Working with low-level HTTP libraries, the client needs to create a client instance and a request object, execute the request, interpret the response, map the response to domain objects, and handle any exceptions that may be thrown along the way. And all of this boilerplate is repeated, regardless of what HTTP request is sent.
- To avoid such boilerplate code, Spring provides RestTemplate. Just as JDBCTemplate handles the ugly parts of working with JDBC, RestTemplate frees you from dealing with the tedium of consuming REST resources.

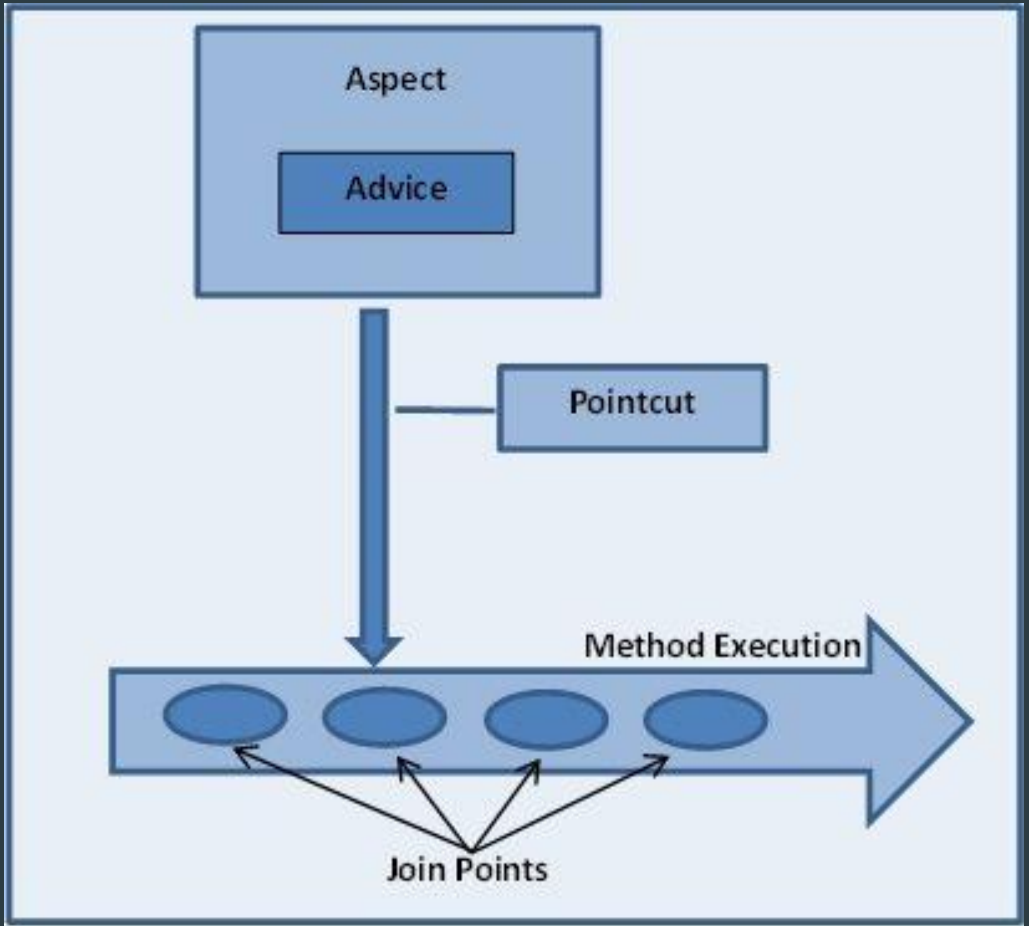
Session Content (23)

- Aspect-Oriented Programming Overview
  - Advice Type
  - Pointcut Expressions
  - Ordering Aspect
- Spring Boot Caching
  - Caching Introduction
  - Types of cache
  - Spring boot cache annotations
  - Integration with cache engine (Redis)

Spring AOP

- AOP **(Aspect-Oriented Programming)** is a programming pattern that increases modularity by allowing the separation of the **cross-cutting concern**. These cross-cutting concerns are different from the main business logic. We can add additional behavior to existing code without modification of the code itself.
- AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations. It makes easy to maintain code in the present and future as well.
- Aspect: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using ***@Aspect*** annotation.
- Join Point: A join point is a specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
- Advice: Advices are actions taken for a particular join point.
- Pointcut: Pointcut is expressions that are matched with join points to determine whether advice needs to be executed or not.

- Target Object: They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object.
- AOP proxy: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes.
- Weaving: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.



AOP Advice Types

- Before Advice: These advices runs before the execution of join point methods. We can use ***@Before*** annotation to mark an advice type as Before advice.
- After (finally) Advice: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using ***@After*** annotation.
- After Returning Advice: Sometimes we want advice methods to execute only if the join point method executes normally. We can use ***@AfterReturning*** annotation to mark a method as after returning advice.
- After Throwing Advice: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use ***@AfterThrowing*** annotation for this type of advice.
- Around Advice: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use ***@Around*** annotation to create around advice methods.

Spring Boot Starter AOP

- Spring Boot Starter AOP is a dependency that provides Spring AOP and AspectJ. Where AOP provides basic AOP capabilities while the AspectJ provides a complete AOP framework.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

- Enable AOP configuration in Spring applications.

```
@Configuration
@EnableAspectJAutoProxy
```

Aspects Ordering

- When you have multiple aspects in your application and they are can be applied on a certain method, When there’s more than one aspect applied to the same join point, the precedence/order of the aspects will not be determined unless you have explicitly specified it using either @Order annotation or org.springframework.core.Ordered interface. In this example, we will see an example of ordered aspects.
- **The *@Order*** annotation defines the sorting order of an annotated component or bean.
- It has an optional value argument which determines the order of the component; the default value is *Ordered.LOWEST_PRECEDENCE*. This marks that the component has the lowest priority among all other ordered components.
- Similarly, the value *Ordered.HIGHEST_PRECEDENCE* can be used for overriding the highest priority among components.
- The *@Order* annotation when used with the AspectJ execution order. It means the highest order advice will run first.

Spring Boot Caching

What is caching?

- Caching is a mechanism to enhance the performance of a system. It is a temporary memory that lies between the application and the persistent database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Why we need caching?

- Caching of frequently used data in application is a very popular technique to increase performance of application. With caching, we store such frequently accessed data in memory to avoid hitting the costly backends every time when user requests the data. Data access from memory is always faster in comparison to fetching from storage like database, file system or other service calls.

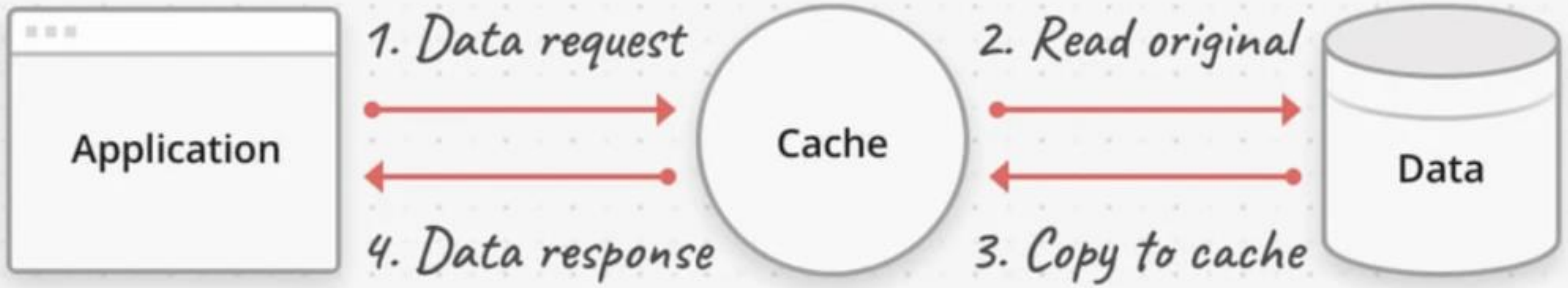
What data should be cached?

- List of products available in an e-commerce store
- Any Master data which is not frequently changed
- Any frequently used database read query, where result does not change in each call at least for a specific period.

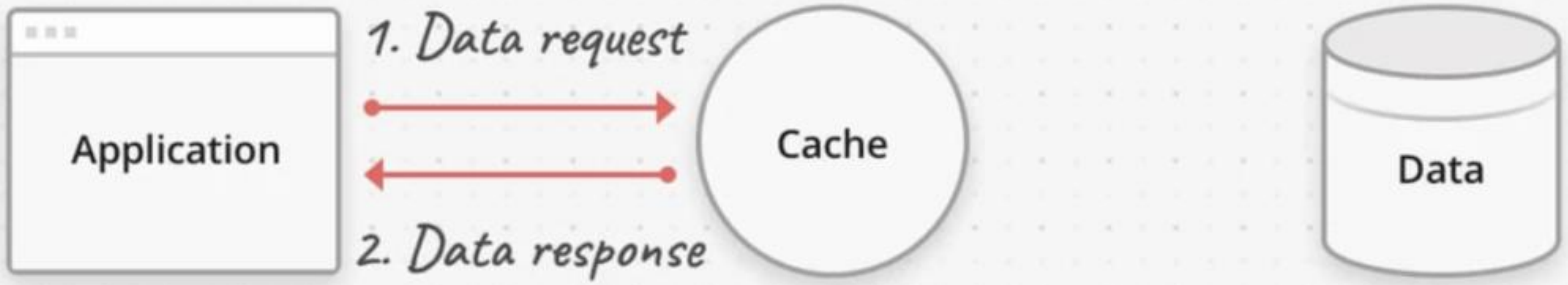
Types of cache

- In-memory caching
- Database caching
- Web server caching
- CDN caching

Cache Miss



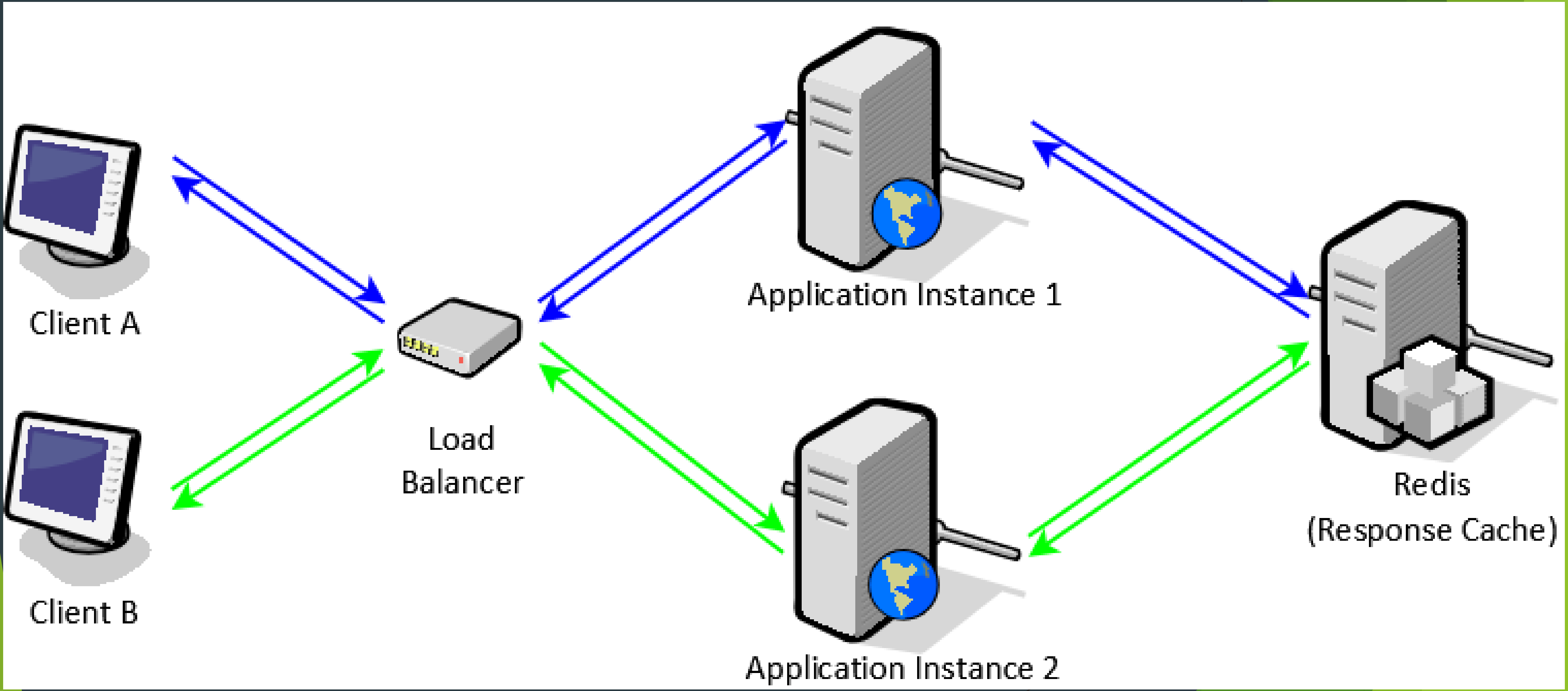
Cache Hit



Spring boot cache annotations

- Spring framework provides **cache abstraction API** for different cache providers. The usage of the API is very simple.
- @EnableCaching : It enables Spring's annotation-driven cache management capability.
- **@Cacheable :** It is used on the method level to let spring know that the response of the method are cacheable.
- @CachePut : Sometimes we need to manipulate the caching manually to put (update) cache before method call, The method will always be executed and its result placed into the cache (according to the @CachePut options).
- @CacheEvict : It is used when we need to evict (remove) the cache previously loaded of master data. When **CacheEvict** annotated methods will be executed, it will clear the cache.
- @Caching : This annotation is required when we need both CachePut and CacheEvict at the same time.
- Spring boot provides integration with following cache providers. Spring boot does the auto configuration with default options if those are present in class path and we have enabled cache by @EnableCaching in the spring boot application (EhCache 2.x, Hazelcast, Caffeine, Redis).

Redis Cache With Spring boot



Session Content (25)

- Lombok
- Data Transfer Object (DTO)
- Mapstruct
- Spring boot profiles

Lombok

- https://projectlombok.org/

Mapstuct

- https://mapstruct.org/
- https://mapstruct.org/documentation/dev/reference/html/
- https://github.com/mapstruct/mapstruct-examples

Spring Boot Profiles

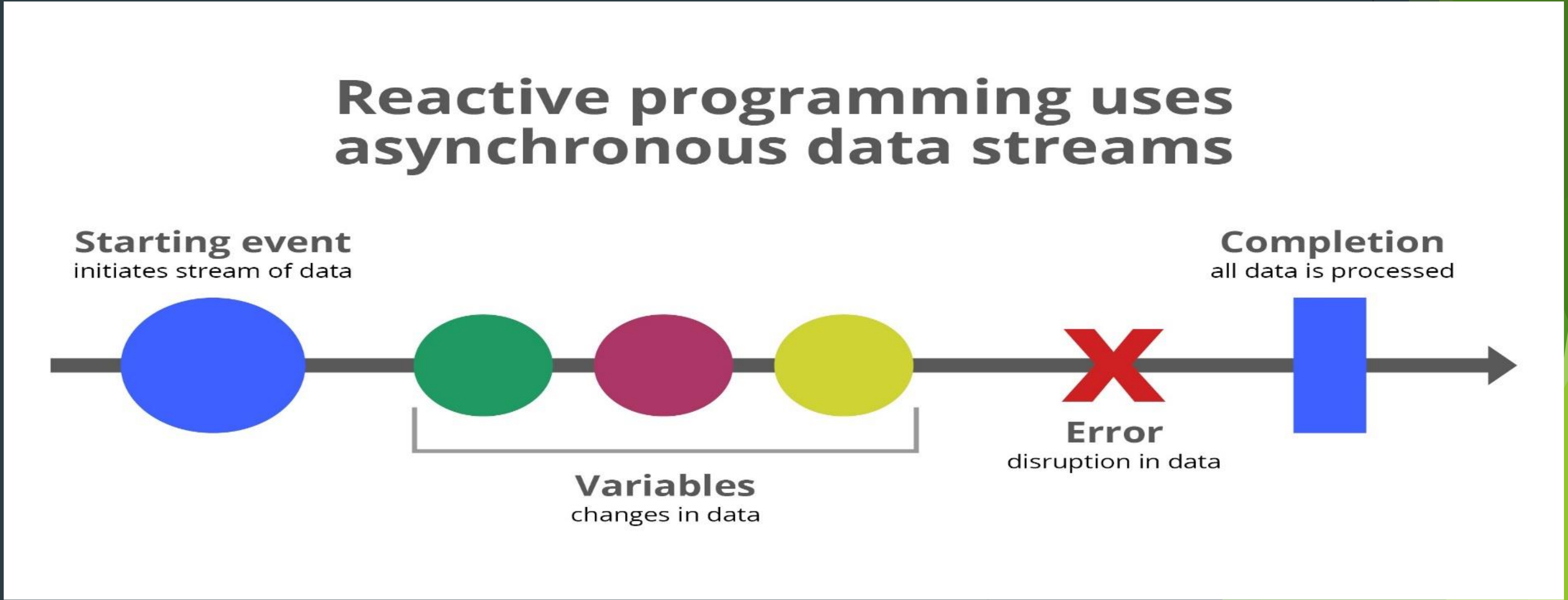
- https://dzone.com/articles/spring-boot-profiles-1

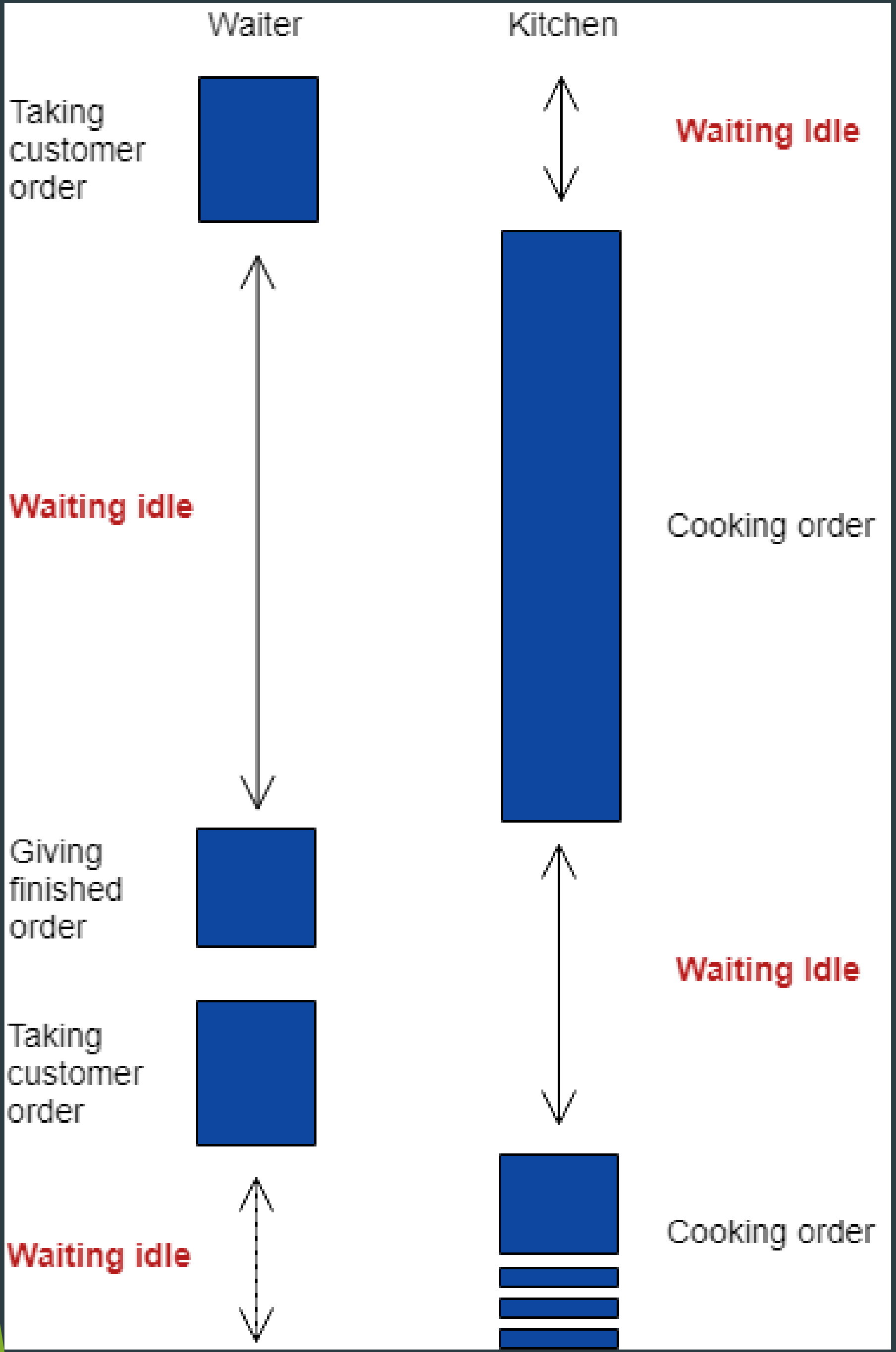
Session Content (26)

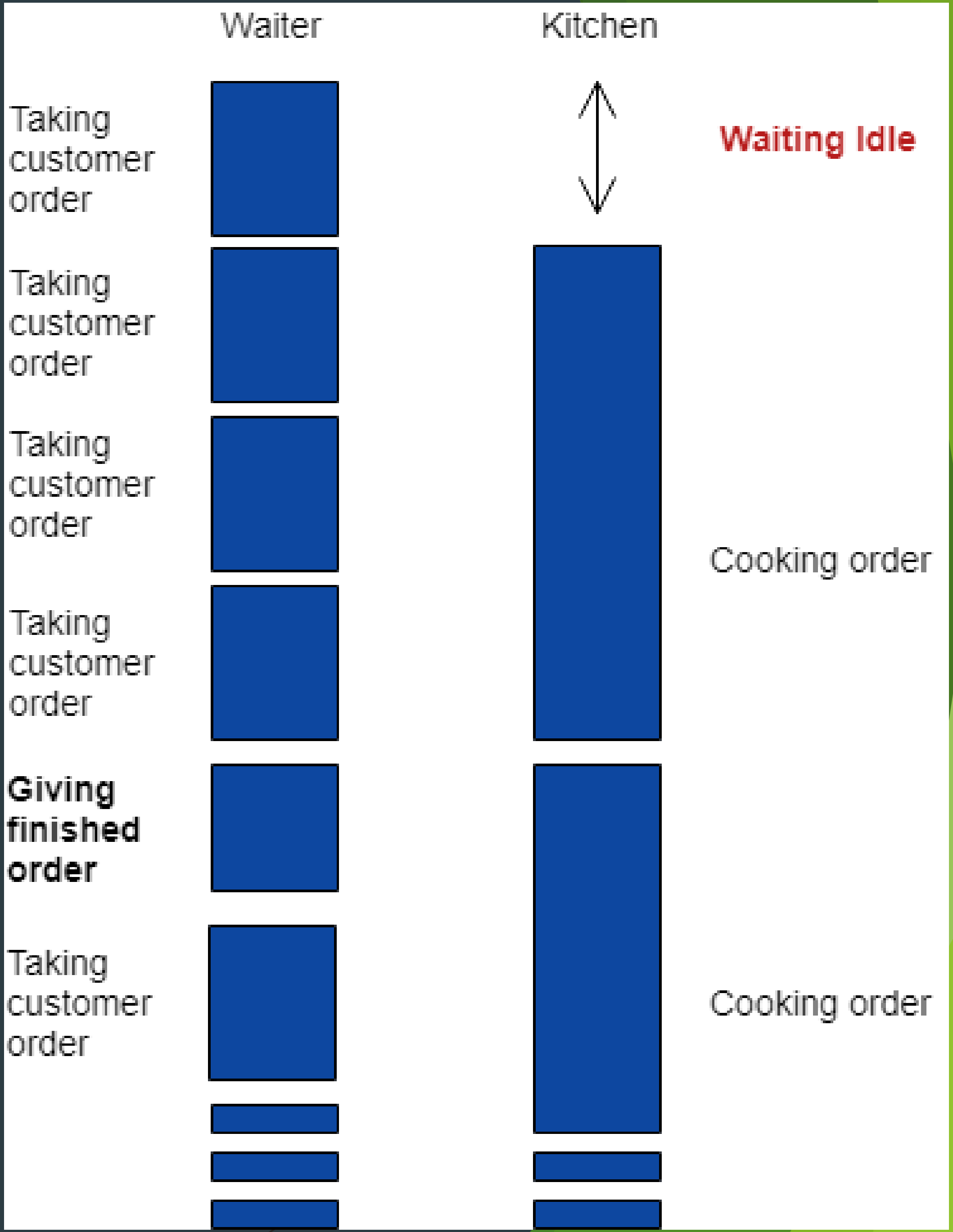
- Reactive Programming
- Blocking vs non-blocking
- Spring WebFlux
- Consuming REST APIs reactively

Reactive Programming

- Reactive programming is a programming paradigm that promotes an asynchronous, non-blocking, event-driven approach to data processing.
- The term, “reactive,” refers to programming models that are built around reacting to changes. It is build around publisher-subscriber pattern (observer pattern). In reactive style of programming, we make a request for resource and start performing other things. When the data is available, we get the notification along with data inform of call back function. In callback function, we handle the response as per application/user needs.

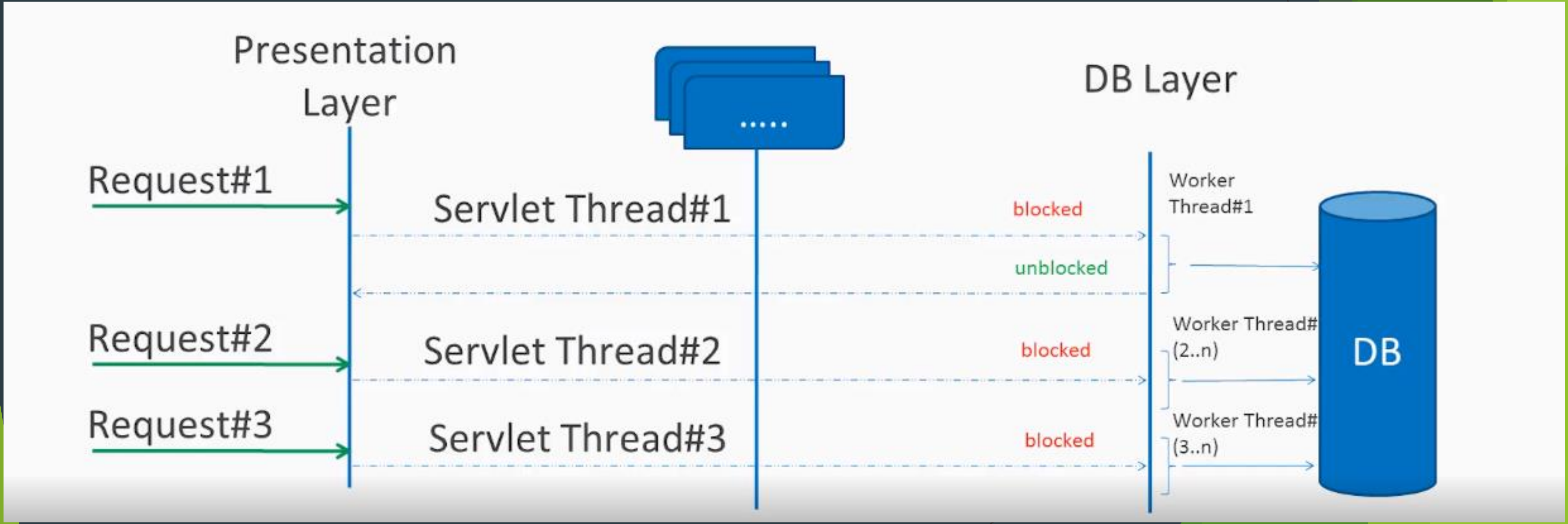



Waiter
Kitchen
Taking customer order
Waiting Idle
Waiting idle
Cooking order
Giving finished order
Waiting Idle
Taking customer order
Waiting idle
Cooking order


Waiter
Kitchen
Taking customer order
Waiting Idle
Taking customer order
Taking customer order
Cooking order
Taking customer order
Giving finished order
Taking customer order
Cooking order

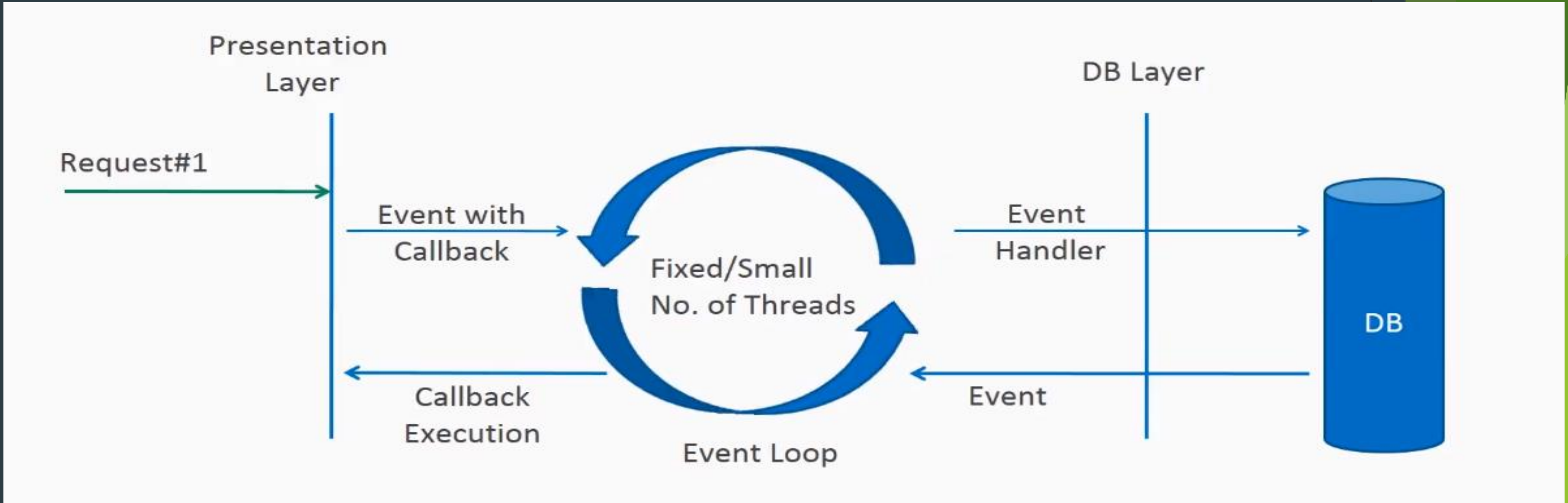
Blocking request processing

- In traditional MVC applications, when a request come to server, a servlet thread is created. It delegates the request to worker threads for I/O operations such as database access etc. During the time worker threads are busy, servlet thread (request thread) remain in waiting status and thus it is blocked. It is also called **synchronous request processing**.



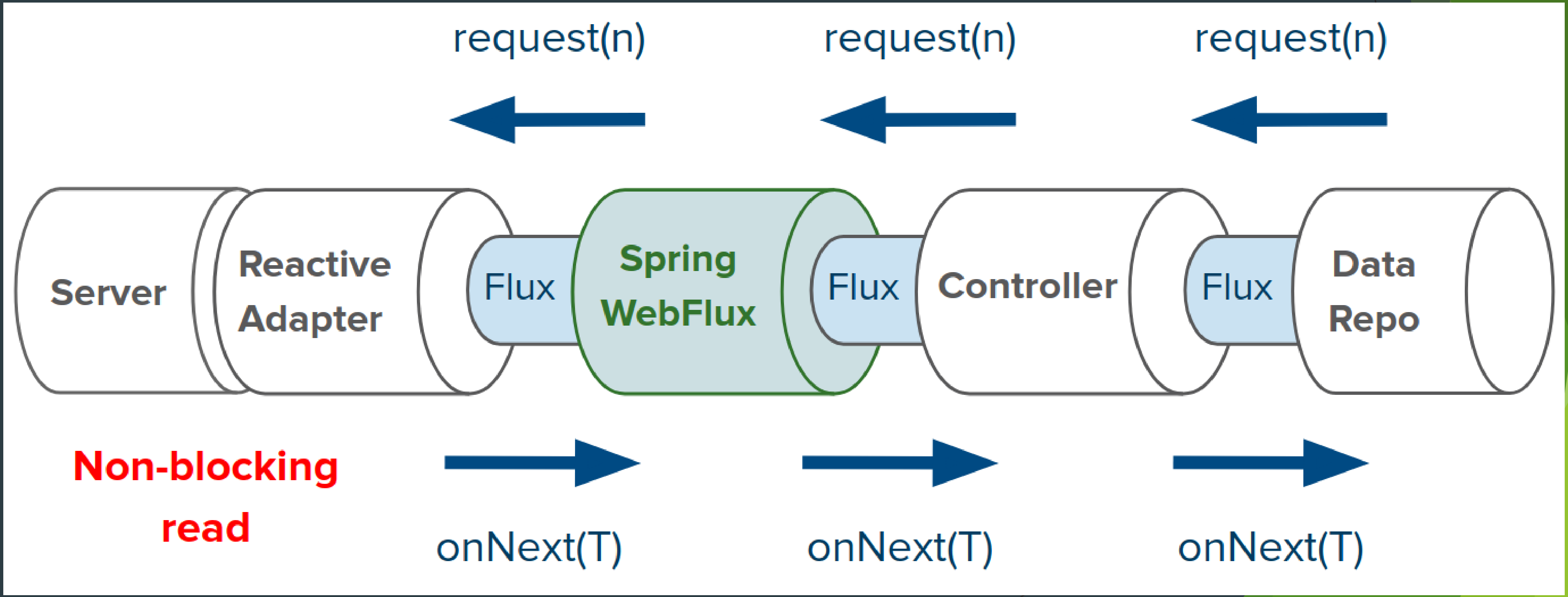
Non-blocking request processing

- In non-blocking or asynchronous request processing, no thread is in waiting state. There is generally only one request thread receiving the request.
- All incoming requests come with a event handler and call back information. Request thread delegates the incoming requests to a thread pool (generally small number of threads) which delegate the request to it's handler function and immediately start processing other incoming requests from request thread.
- When the handler function is complete, one of thread from pool collect the response and pass it to the call back function.



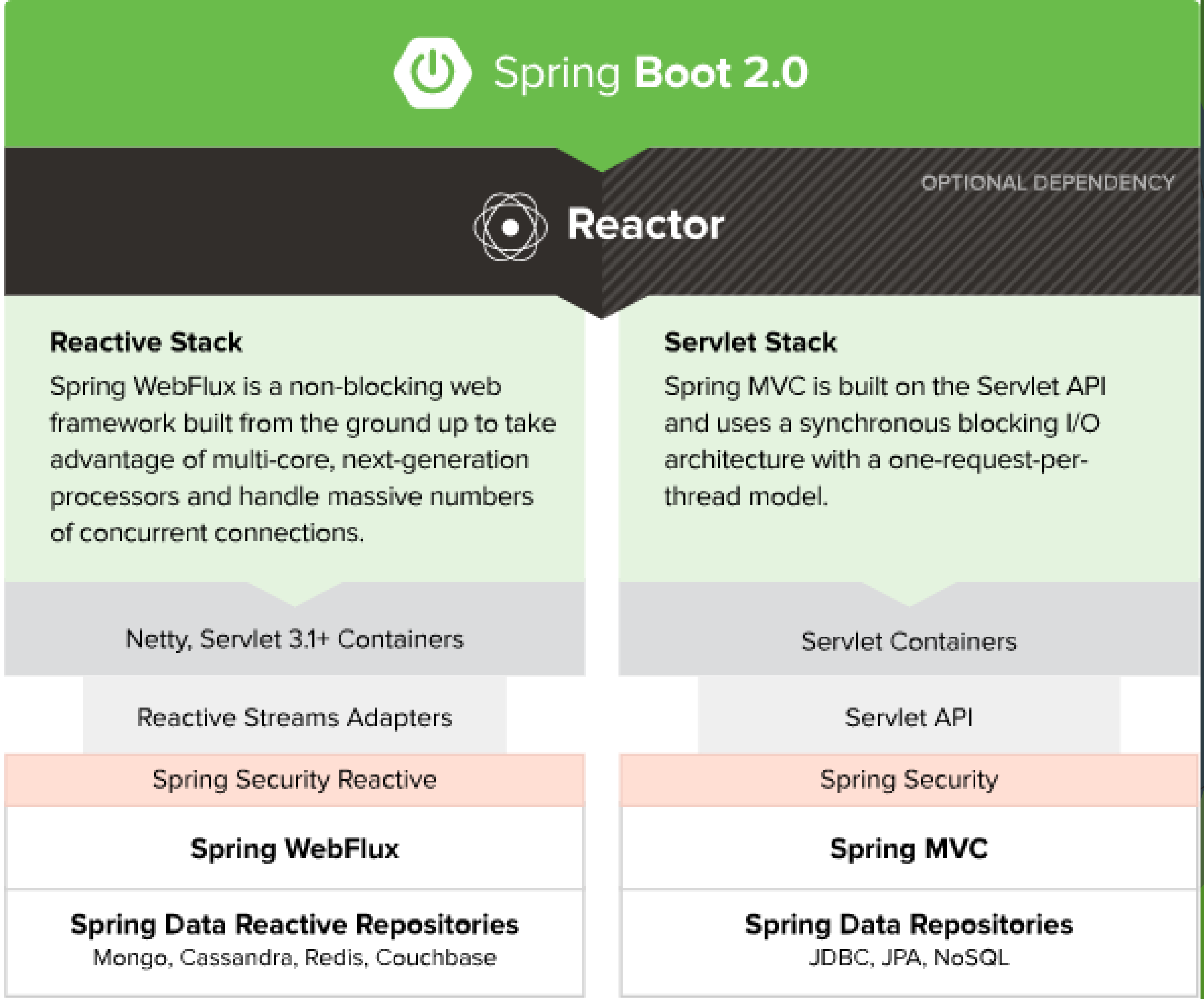
When to use reactive programming

- Reactive web programming is great for applications that have streaming data, and clients that consume it and stream it to their users. It is not great for developing traditional CRUD applications. If you're developing the next *Facebook* or *Twitter* with lots of data, a reactive API might be just what you're looking for.



What is Spring WebFlux

- Spring WebFlux is parallel version of Spring MVC and supports fully non-blocking reactive streams. It support the back pressure concept and uses Netty as inbuilt server to run reactive applications. If you are familiar with Spring MVC programming style, you can easily work on webflux also.
- Spring webflux uses project reactor as reactive library. Reactor is a Reactive Streams library and, therefore, all of its operators support non-blocking back pressure. It is developed in close collaboration with Spring.
- Spring WebFlux heavily uses two publishers :
  - Mono: Returns 0 or 1 element.
  - Flux: Returns 0…N elements. A Flux can be endless, meaning that it can keep emitting elements forever. Also it can return a sequence of elements and then send a completion notification when it has returned all of its elements.
- In Spring WebFlux, we call reactive APIs/functions that return monos and fluxes and your controllers will return monos and fluxes. When you invoke an API that returns a mono or a flux, it will return immediately. The results of the function call will be delivered to you through the mono or flux when they become available.
- To build a truly non-blocking application, we must aim to create/use all of its components as non-blocking i.e. client, controller, middle services and even the database. If one of them is blocking the requests, our aim will be defeated.



Reactive Stack

Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.

Servlet Stack

Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.

Reactive Stack	Servlet Stack
Netty, Servlet 3.1+ Containers	Servlet Containers
Reactive Streams Adapters	Servlet API
Spring Security Reactive	Spring Security
Spring WebFlux	**Spring MVC**
Spring Data Reactive Repositories Mongo, Cassandra, Redis, Couchbase	**Spring Data Repositories** JDBC, JPA, NoSQL

Consuming REST APIs reactively

- Using WebClient is quite different from using RestTemplate. Rather than have several methods to handle different kinds of requests, WebClient has a fluent builderstyle interface that lets you describe and send requests. The general usage pattern for working with WebClient is:-
  - Create an instance of WebClient (or inject a WebClient bean)
  - Specify the HTTP method of the request to send
  - Specify the URI and any headers that should be in the request
  - Submit the request
  - Consume the response

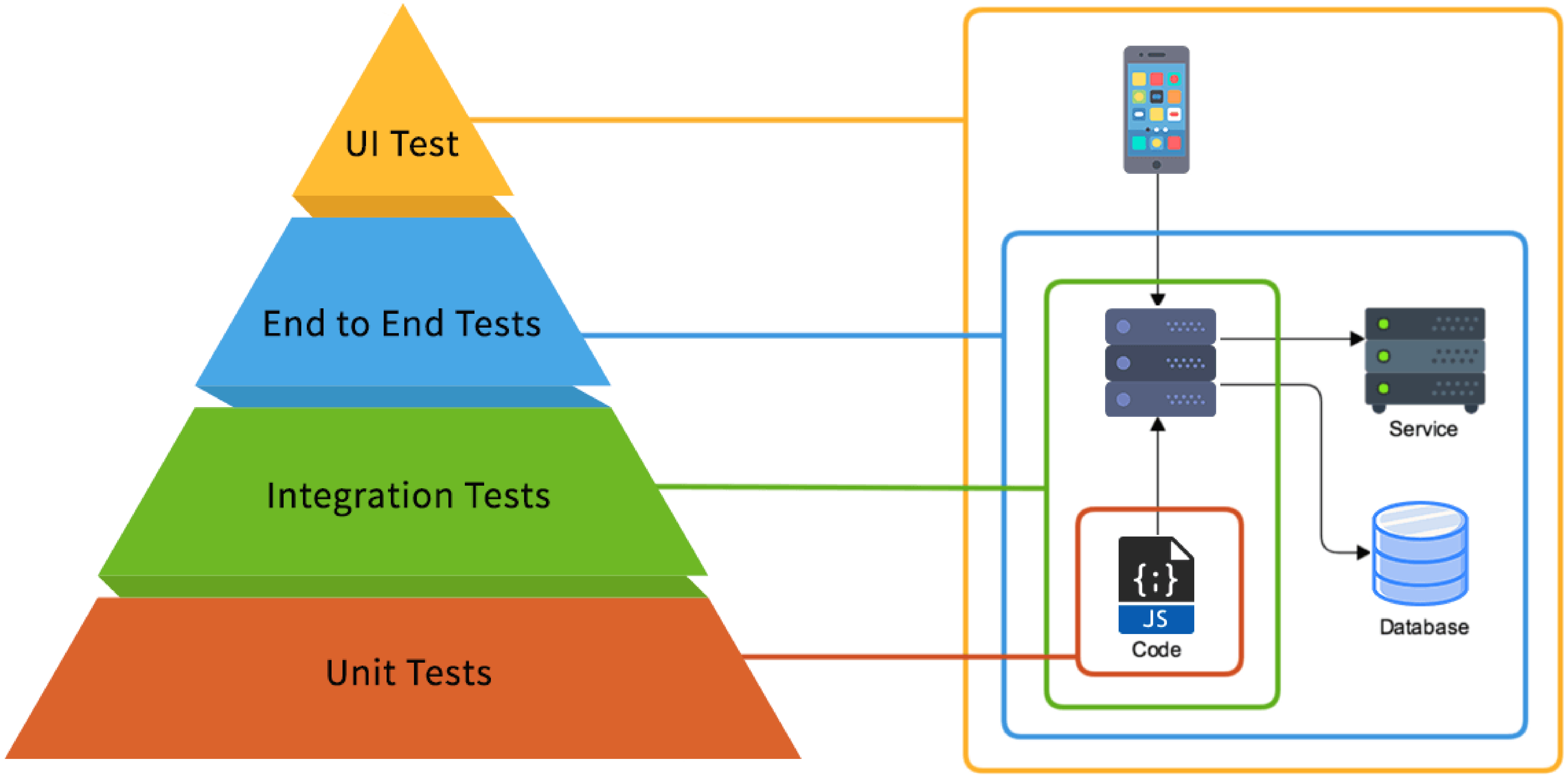
```
Flux<Ingredient> ingredients = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);
```

Session Content (28)

- Unit Test and Integration Test
- Spring Boot Testing
- Mockito

Testing Spring Boot

- Testing is an important part of software development. It helps developers verify the correctness of the functionality.
- JUnit and TestNG are two of the most popular testing libraries used in Java projects.
- Test Driven Development (TDD) is a popular development practice where you write tests first and write just enough production code to pass the tests.
- You write various types of tests, such as unit tests, integration tests, performance tests, etc.
- Unit tests focus on testing one component in isolation, whereas integration tests verify the behavior of a feature, which could possibly involve multiple components.
- While doing integration testing, you may have to mock the behavior of dependent components such as third-party web service classes, database method invocations, etc.
- There are mocking libraries like **Mockito, PowerMock, and jMock**, for mocking the object's behavior.
- Spring Boot provides the @SpringBootTest annotation which we can use to create an application context containing all the objects we need for all of the above test types.


UI Test
End to End Tests
Integration Tests
Unit Tests
Service
Database
JS
Code

The Spring Boot Test starter

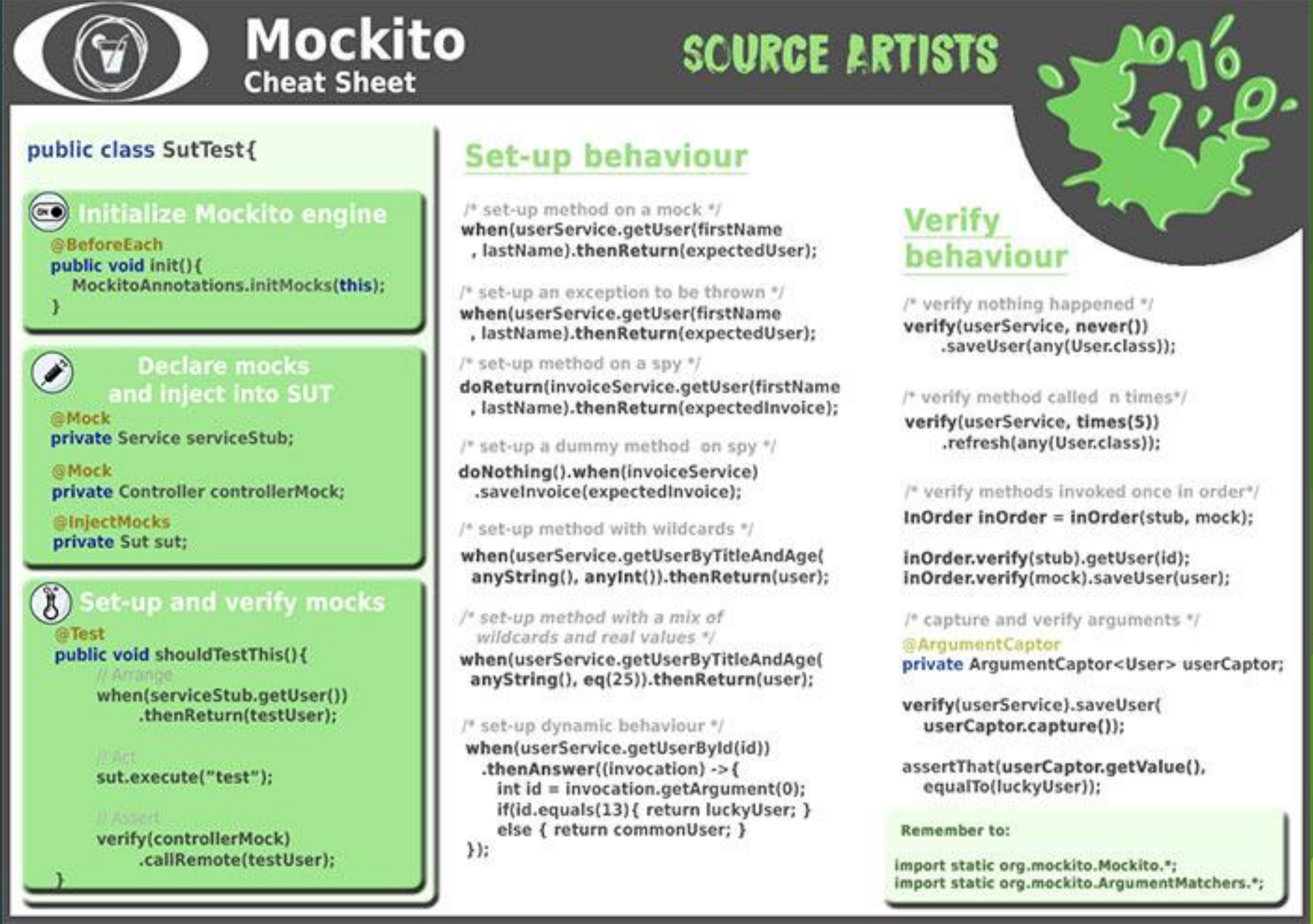
- The Spring Boot Test starter pulls in the JUnit, Spring Test, and Spring Boot Test modules, along with the following most commonly used mocking and asserting libraries:
- JUnit is a unit testing framework for the Java programming language.
- Mockito is a Java mocking framework.
- AssertJ is a collection of utility methods that support asserting conditions in tests.
- Hamcrest is a matcher/predicates library for data assertion.
- JSONassert is An assertion library for JSON.
- JsonPath XPath for JSON.

Web Environment

- As you need to test the REST endpoint, you start the embedded servlet container by specifying the webEnvironment attribute of @SpringBootTest.
- The default webEnvironment value is WebEnvironment.MOCK, which doesn't start an embedded servlet container.
- You can use various webEnvironment values based on how you want to run the tests.
  - MOCK (default)—Loads a WebApplicationContext and provides a mock servlet environment. It will not start an embedded servlet container. If servlet APIs are not on your classpath, this mode will fall back to creating a regular non-web ApplicationContext.
  - RANDOM_PORT—Loads a ServletWebServerApplicationContext and starts an embedded servlet container listening on a random available port.
  - DEFINED_PORT—Loads a ServletWebServerApplicationContext and starts an embedded servlet container listening on a defined port (server.port).
  - NONE—Loads an ApplicationContext using SpringApplication but does not provide a servlet environment.
- The TestRestTemplate bean will be registered automatically only when @SpringBootTest is started with an embedded servlet container

Mockito

- **What is Mocking?**
- Mocking is a way to test the functionality of a class in isolation. Mocking does not require a database connection or properties file read or file server read to test a functionality. Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.
- **Mockito**
- Mockito facilitates creating mock objects seamlessly. It uses Java Reflection in order to create mock objects for a given interface. Mock objects are nothing but proxy for actual implementations.
- Consider a case of Stock Service which returns the price details of a stock. During development, the actual stock service cannot be used to get real-time data. So we need a dummy implementation of the stock service. Mockito can do the same very easily, as its name suggests.



Mockito
Cheat Sheet

SOURCE ARTISTS

```
public class SutTest{
```

Initialize Mockito engine

```
@BeforeEach
public void init(){
   MockitoAnnotations.initMocks(this);
}
```

Declare mocks and inject into SUT

```
@Mock
private Service serviceStub;

@Mock
private Controller controllerMock;

@InjectMocks
private Sut sut;
```

Set-up and verify mocks

```
 @Test
 public void shouldTestThis(){
        // Arrange
        when(serviceStub.getUser())
              .thenReturn(testUser);

        // Act
        sut.execute("test");

        // Assert
        verify(controllerMock)
              .callRemote(testUser);
 }
```

Set-up behaviour

```
/* set-up method on a mock */
when(userService.getUser(firstName
 , lastName).thenReturn(expectedUser);

/* set-up an exception to be thrown */
when(userService.getUser(firstName
 , lastName).thenReturn(expectedUser);
/* set-up method on a spy */
doReturn(invoiceService.getUser(firstName
 , lastName).thenReturn(expectedInvoice);

/* set-up a dummy method  on spy */
doNothing().when(invoiceService)
  .saveInvoice(expectedInvoice);

/* set-up method with wildcards */
when(userService.getUserByTitleAndAge(
 anyString(), anyInt()).thenReturn(user);

/* set-up method with a mix of
  wildcards and real values */
when(userService.getUserByTitleAndAge(
 anyString(), eq(25)).thenReturn(user);

/* set-up dynamic behaviour */
when(userService.getUserById(id))
  .thenAnswer((invocation) ->{
     int id = invocation.getArgument(0);
     if(id.equals(13){ return luckyUser; }
     else { return commonUser; }
});
```

Verify behaviour

```
/* verify nothing happened */
verify(userService, never())
      .saveUser(any(User.class));

/* verify method called  n times*/
verify(userService, times(5))
      .refresh(any(User.class));

/* verify methods invoked once in order*/
InOrder inOrder = inOrder(stub, mock);

inOrder.verify(stub).getUser(id);
inOrder.verify(mock).saveUser(user);

/* capture and verify arguments */
@ArgumentCaptor
private ArgumentCaptor<User> userCaptor;

verify(userService).saveUser(
   userCaptor.capture());

assertThat(userCaptor.getValue(),
   equalTo(luckyUser));
```

Remember to:

```
import static org.mockito.Mockito.*;
import static org.mockito.ArgumentMatchers.*;
```

Session Content (30)

- Securing Web Applications
- Spring Boot Security
- Authentication & Authorization
- Session VS Token based authentication

Securing Web Applications

- Security is an important aspect of software application design. It ensures that only those who have authority to access the secured resources can do so.
- When it comes to securing an application, two primary things you'll need to take care of are authentication and authorization.
- Authentication refers to the process of verifying the user, which is typically done by asking for credentials.
- Authorization refers to the process of verifying whether or not the user is allowed to do a certain activity.
- Spring Security is a powerful and flexible security framework for securing Java-based web applications.
- Even though Spring Security is commonly used with Spring-based applications, you can use it to secure non-Spring-based web applications too.

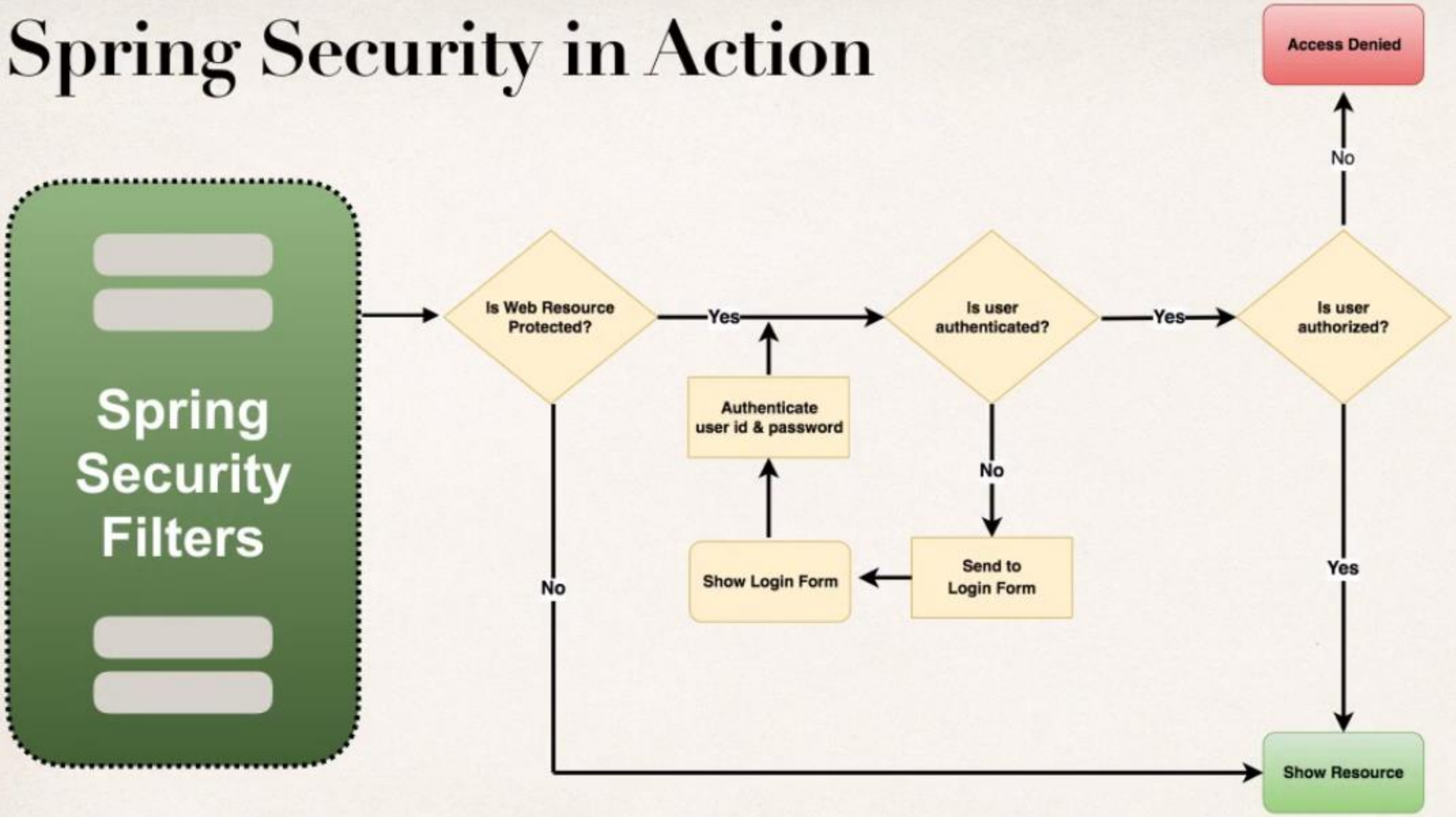
Spring Boot Security

- Spring Security is a framework for securing Java-based applications at various layers with great flexibility and customizability.
- Spring Security provides authentication and authorization support against ***database*** authentication, ***LDAP***, ***form*** authentication, ***JA-SIG* central** authentication service, Java Authentication and Authorization Service ***(JAAS)***, and many more.
- Spring Security provides support for dealing with common attacks like **CSRF**, **XSS**, and session fixation protection, with minimal configuration.
- Spring Security can be used to secure the application at various layers, such as web URLs, service layer methods, etc.
- Adding the Spring Security Starter (spring-boot-starter-security) to an Spring Boot application will:-
  - Enable HTTP basic security
  - Register the AuthenticationManager bean with an in-memory store and a single user
  - Ignore paths for commonly used static resource locations (such as /css/**, /js/**, /images/**, etc.)
  - Enable common low-level features such as XSS, CSRF, caching, etc.

Security Concepts

- Authentication
  - Check user id and password with credentials stored in app / db
- Authorization
  - Check to see if user has an authorized role

Spring Security in Action



Declarative Security

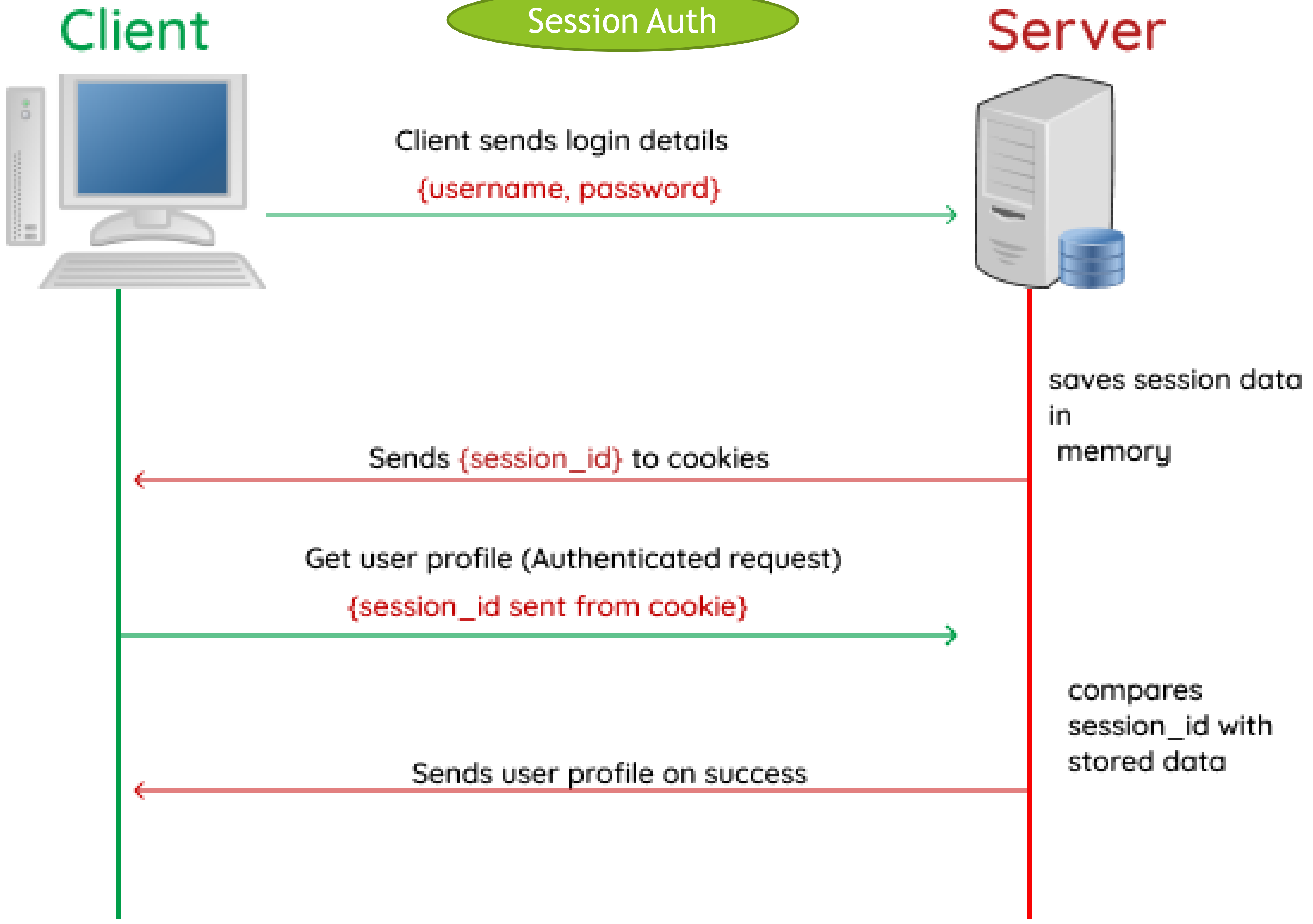
- Define application's security constraints in configuration
    - All Java config (@Configuration, no xml)
    - or Spring XML config
- Provides separation of concerns between application code and security

Authentication and Authorization

- In-memory
- JDBC
- LDAP
- Custom / Pluggable
- *others …*

```
users
passwords
roles
```

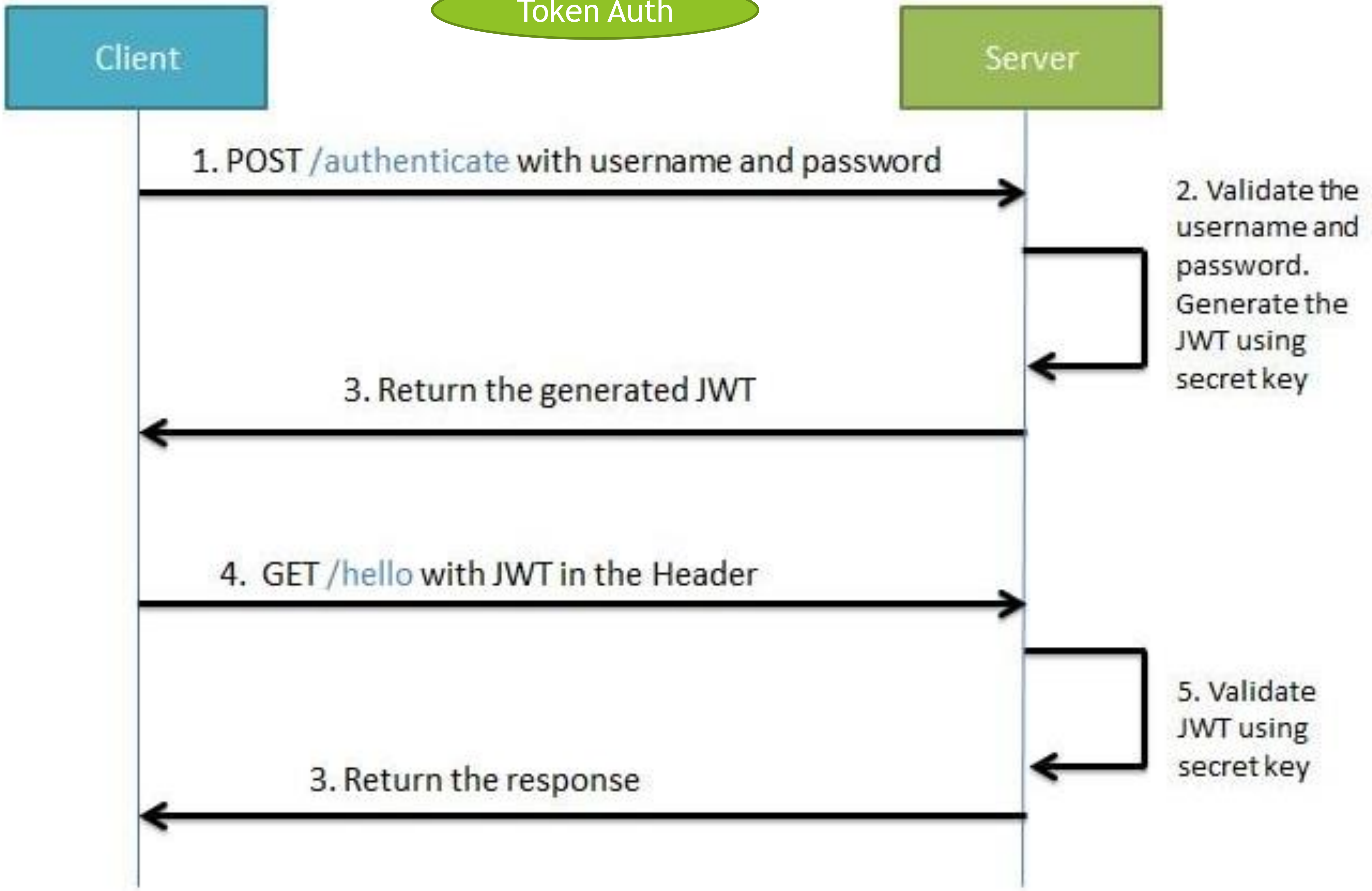
Session Auth



Client
Server
Client sends login details
{username, password}
saves session data in memory
Sends {session_id} to cookies
Get user profile (Authenticated request)
{session_id sent from cookie}
compares session_id with stored data
Sends user profile on success

Session based authentication

Token Auth


Client
Server
1. POST /authenticate with username and password
2. Validate the username and password. Generate the JWT using secret key
3. Return the generated JWT
4. GET /hello with JWT in the Header
5. Validate JWT using secret key
3. Return the response

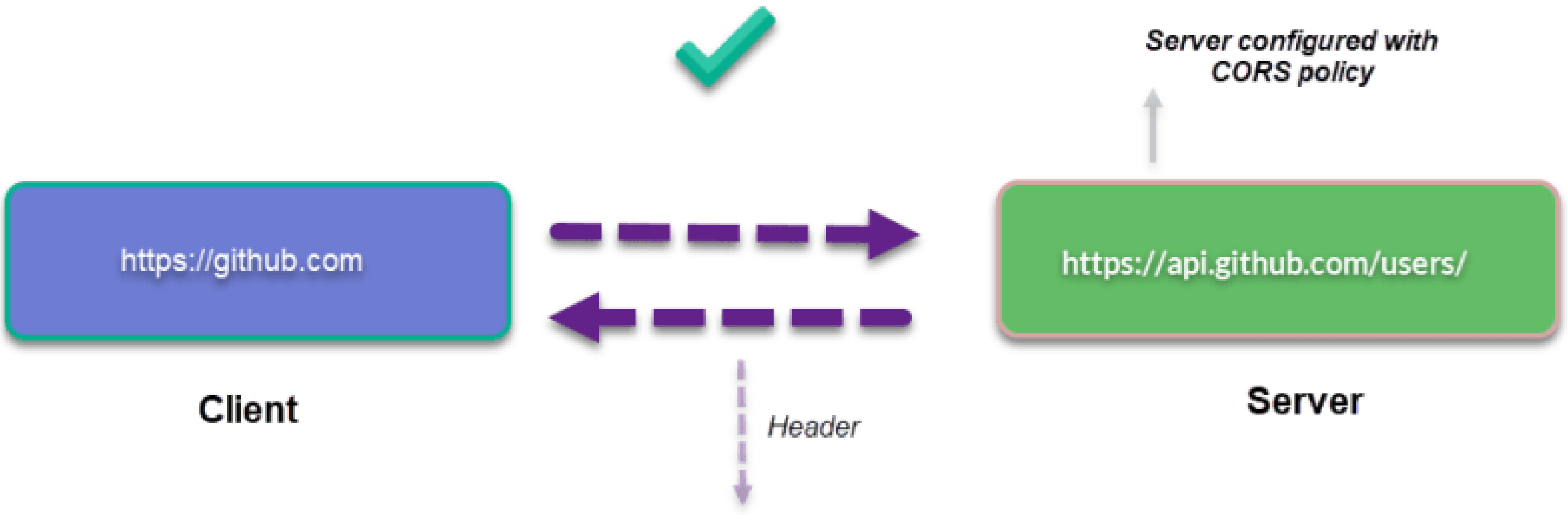
Method-Level Security

- Spring Security provides method-level security using the @Secured annotation.
- It also supports the JSR-250 security annotation @RolesAllowed.
- From version 3.0, Spring Security has provided an expression-based security configuration using the @PreAuthorize and @PostAuthorize annotations, which provides more fine-grained control.
- You can enable method-level security using the @EnableGlobalMethodSecurity annotation on any configuration class
- Similarly, you can secure service-layer methods using the @Secured, @PreAuthorize, or @RolesAllowed annotations.
- You can use the Spring Expression Language (SpEL) to define the security expressions as follows:
  - hasRole(role): Returns true if the current user has the specified role.
  - hasAnyRole(role1,role2): Returns true if the current user has any of the supplied roles.
  - isAnonymous(): Returns true if the current user is an anonymous user.
  - isAuthenticated(): Returns true if the user is not anonymous.
  - isFullyAuthenticated(): Returns true if the user is not an anonymous or Remember-Me user

Spring Boot - CORS Support

- Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.
- For example, your web application is running on 8080 port and by using JavaScript you are trying to consuming RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.
- Two requirements are needed to handle this issue –
  - RESTful web services should support the Cross-Origin Resource Sharing.
  - RESTful web service application should allow accessing the API(s) from the 8080 port.
- We need to set the origins for RESTful web service by using **@CrossOrigin** annotation for the controller method.
- This @CrossOrigin annotation supports specific REST API, and not for the entire application.

How CORS Works ?


Server configured with CORS policy
https://github.com
https://api.github.com/users/
Client
Server
Header

Access-Control-Allow-Origin : https://github.com