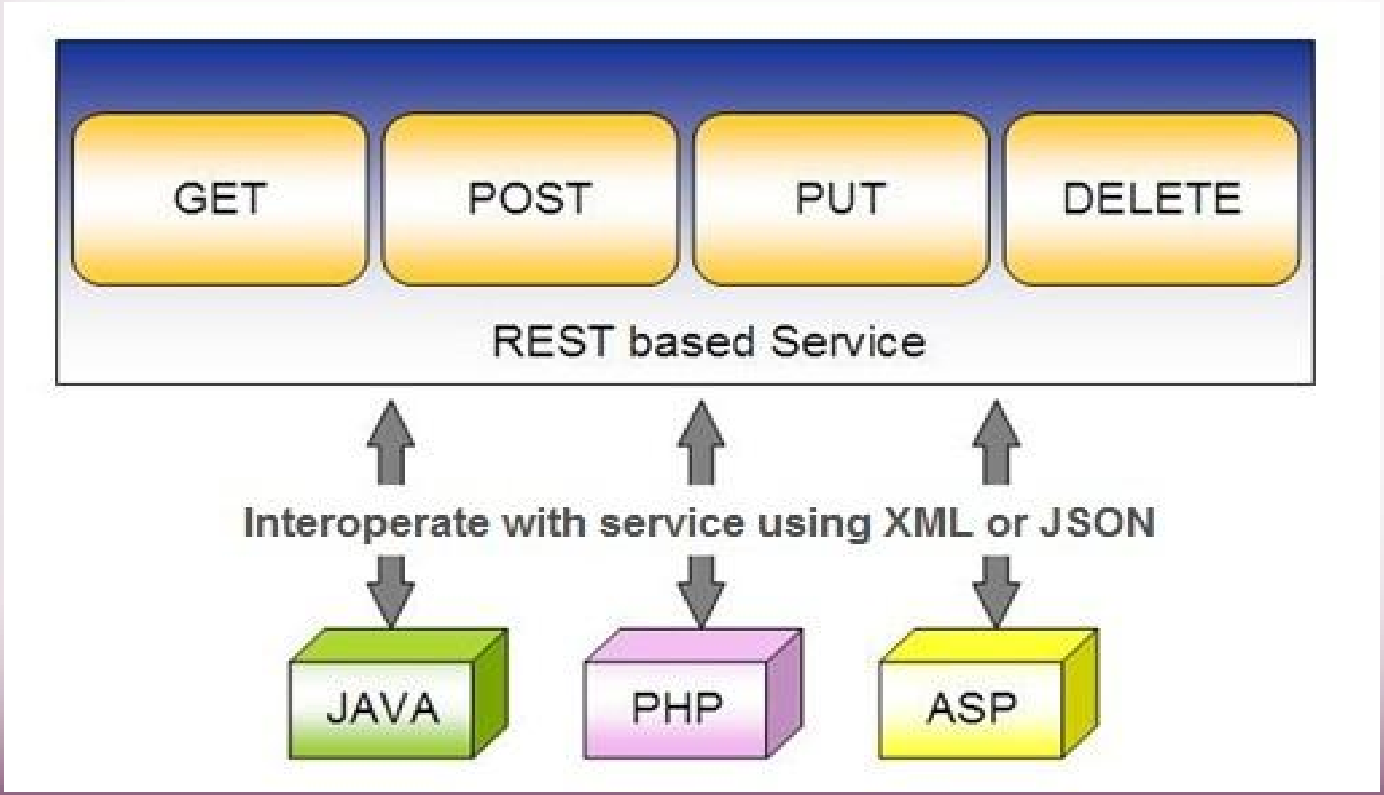
REST SERVICE USING SPRING DATA JPA

Spring RESTful Services

- *REST* does not require the client to know anything about the structure of the API.
- Server needs to provide whatever information the client needs to interact with the service.



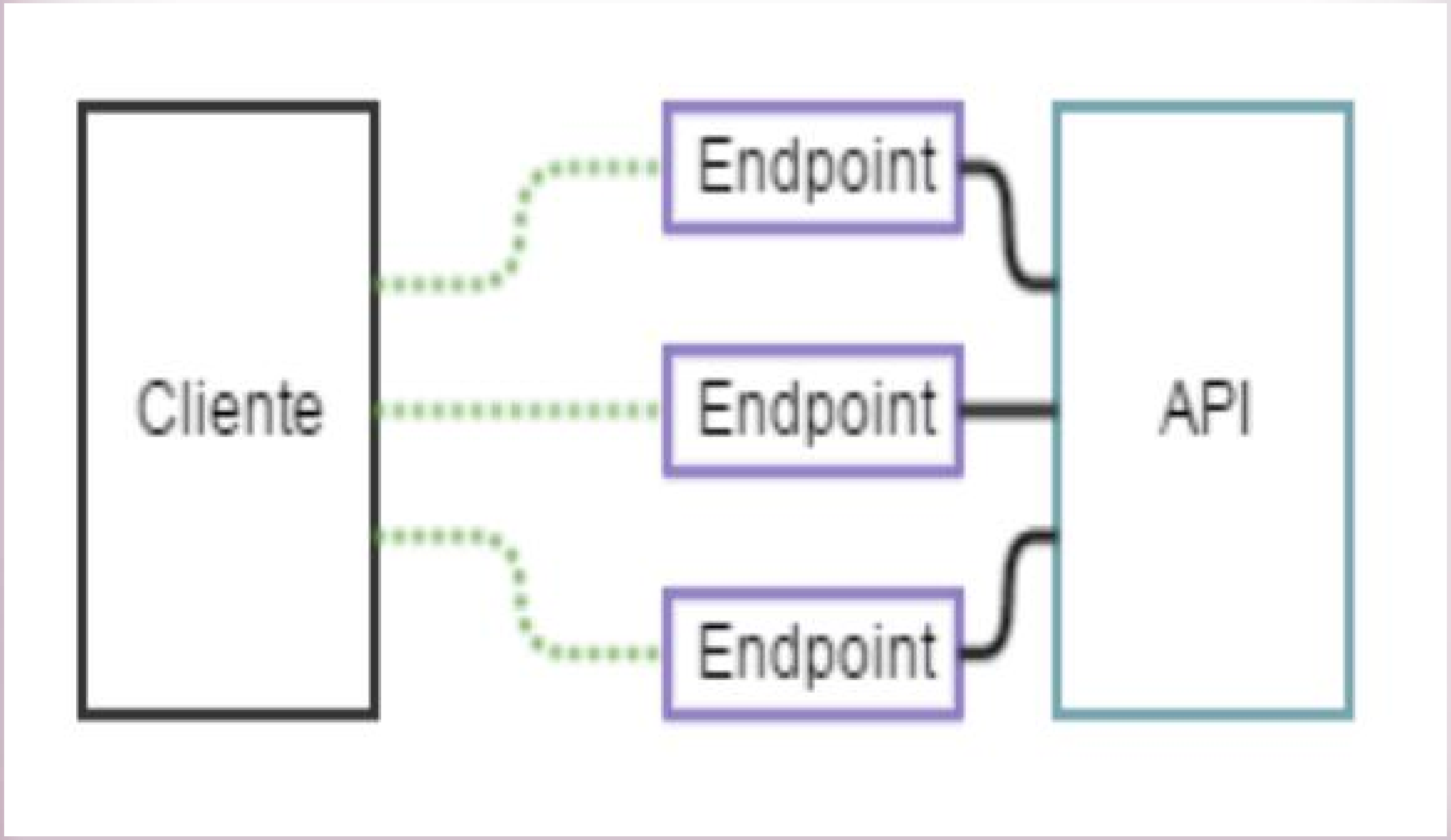
Using appropriate Request Methods

- **GET :**
  - Should not update anything.
  - Should be idempotent (same result in multiple calls).
  - Possible Return Codes 200 (OK) + 404 (NOT FOUND) +400 (BAD REQUEST)
- **POST :**
  - Should create new resource.
  - Ideally return JSON with link to newly created resource.
  - Same return codes as get possible.
  - In addition : Return code 201 (CREATED) is possible.

Using appropriate Request Methods

- **PUT :**
  - Update a known resource. ex: update client details.
  - Possible Return Codes : 200(OK)
- **DELETE :**
  - Used to delete a resource.
  - Possible Return Codes : 200(OK) or a 204 - no content

API and Endpoint



SPRING DATA REST WITH JPA

Spring RESTful Services

- A RESTful architecture may expose multiple representations of a resource.

- From Spring 4   @RestController annotation is added to controller
  - A combination of @Controller and @ResponseBody

- A Rest Controller method returns a domain object instead of a view.

REST Stereotypes

- **@RestController**
  - An implicit @ResponseBody is being added to the methods.
  - Allows Spring to render the returned HttpEntity and its payload, directly to the response.

- **@GetMapping**
  - To map HTTP GET requests onto specific handler methods.
  - *Composed annotation* for @RequestMapping(method = RequestMethod.GET).

REST Stereotypes

- **@PathVariable**
  - Indicates that the Method parameter should be bound to a URI template variable.

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

public String getById(@PathVariable String id) { }

- **@PostMapping**
  - Combined shortcut for @RequestMapping(method = RequestMethod.POST).

@PostMapping("/members“)

public void addMember(**@RequestBody Member** member) { }

@Repository

- There are three repository in Spring Data
- ***CrudRepository***
  - Extends Repository
  - provides CRUD functions
- ***PagingAndSortingRepository***
  - Extends CrudRepository
  - provides methods to do pagination and sort records
- ***JpaRepository***
  - Extends PagingandSortingRepository
  - Provides Methods such as flushing the persistence context and delete records in a batch
  - Querying methods return List's instead of Iterable's

Creating Spring Data Application

1. Create a repository interface and extend one of the repository interfaces provided by Spring Data.
2. If required add custom query methods to the created repository interface
3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring.

- Need **NOT** create an implementation class
  - *Spring will automatically create its implementation class at runtime.*
- The Repository class will be auto detected if suitably placed in the scan path

CrudRepository Interface

- Extends Repository Interface and has the following Methods
  - **<S extends T> S save(S entity)**
  - <S extends T> Iterable<S> saveAll(Iterable<S> entities);
  - **Optional<T> findById(ID primaryKey)**
  - Iterable<T> findAll()
  - **void delete(T entity)**
  - long count()
  - **boolean existsById(ID id);**

PagingAndSortingRepository

- Extends CrudRepository Interface and has the following Methods
- *findAll(Pageable pageable)*
    - *Pageable* object with following properties
        - Page size
        - Current page number
        - Sorting
- *findAll(Sort sort)*
    - *Sort Object with the Property on Which the sorting is to be done*
    - ***Sort.by(propName)***

JpaRepository

- Extends **PagingAndSortingRepository**
- Can also optionally extend **QueryByExampleExecutor**

```
List<T> findAllById(Iterable<ID> ids);

<S extends T> S saveAndFlush(S entity);

void deleteInBatch(Iterable<T> entities);

void deleteAllInBatch();

T getOne(ID id);

void flush();
```

SAMPLE CODE-DEMO

Entity

```
@Entity
@Table(name = "demo_invoice")
@Data
@AllArgsConstructor
@NoArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE)
public class Invoice {
     @Id
     long  id;
    String customerName;
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
     LocalDate invoiceDate;
    double amount;
}
```

Repository

```
@Repository
public interface InvoiceRepository extends
                JpaRepository<Invoice, Long>{



  }
```

Service

```
@Service
public class InvoiceService {

@Autowired
 private  InvoiceRepository repo ;

    public List<Invoice> findAll(){
return repo.findAll();
   }
   public  Invoice  findById(int id) {
   Optional<Invoice> optInvoice=repo.findById(id);
          return optInvoice.orElseThrow(() -> new
                     RunTimeException(“Invoice not found"));
   }
```

Controller

```
@RestController
public class InvoiceController {
@Autowired
 private  InvoiceService service ;

   @GetMapping(path = "/api/v1/invoices")
   public List<Invoice> findAll() {
      List<Invoice> details = service.findAll();
        return details;
}
```

@RequestBody

- Used to map the *HttpRequest* body to a transfer or domain object
  - Spring automatically deserializes the JSON into a Java type
  - The type of *@RequestBody* annotation must correspond to the JSON sent from client-side

- **HttpMessageConverter**
  - Internally Used by Spring to perform Serialization/deserialization
- **@Valid**
  - To perform automatic validation by adding to method argument

Post and Put Mapping

```
@PostMapping(path = "/api/v1/invoices")
public Invoice add(@RequestBody  Invoice  entity
                        ,HttpServletResponse response) {

        response.setStatus(201);
                return this.service.add(entity);
}

  @ResponseStatus(HttpStatus.CREATED)
  @PostMapping(path = "/api/v1/invoices")
  public Invoice add(@RequestBody  Invoice  entity) {
            return this.service.add(entity);
}
```

ResponseEntity

- Represents Complete HTTP response, gives more flexibility
  - May be required in some special cases
  - ***Can be used to send Status Code,Response Body,Location to the resource which was altered***

```
@GetMapping(path = "/api/v1/invoices")
public ResponseEntity<Invoice> getInvoices() {

    HttpHeaders responseHeaders = new HttpHeaders();

return ResponseEntity.ok().headers(responseHeaders).body(dao.findAll());


}
```

Put Mapping

```
@PutMapping(path = "/api/v1/invoice")
public Invoice update(@RequestBody Invoice entity) {
    return this.service.add(entity);
}
@PatchMapping(path = "/api/v1/invoice")
public ResponseEntity<String> updateAmount() {

 return  ResponseEntity.status(HttpStatus.CREATED)
   .body("Updated 
Amount:="+this.service.updateAmount());

}
```

Remove Method in Service

```
 public Invoice remove(Invoice entity) {

  Optional<Invoice> optional = Optional.empty();

if(this.repository.existsById(entity.getId())) {

this.repository.delete(entity);

optional = Optional.of(entity);
}

   return optional;
}
```

Delete Mapping in Controller

```
@DeleteMapping(path = "/api/v1/invoice")

public ResponseEntity<Invoice> remove(@RequestBody Invoice entity) {

Invoice invoice = this.service.removeInvoice(entity).orElseThrow(()-> new
    RuntimeException("Element NOT FOUND"));



        return ResponseEntity.ok().body(invoice);


}
```

@RestControllerAdvice

- Used for Exception handling in Rest API’s
- Used to handle exceptions across the whole application
- Like an interceptor of exceptions thrown by methods annotated with @RequestMapping or one of the shortcuts.
- It’s a combination of @ContollerAdvice and @ResponseBody.

- **@ExceptionHandler**
  - Annotation for handling exceptions in specific handler classes and/or handler methods.

@RestControllerAdvice

```
@RestControllerAdvice
public class ErrorHandler  {
@ExceptionHandler(value = Exception.class)
public ErrorDetails handleAllExceptions(Exception ex,
WebRequest request){

    return   new ErrorDetails(LocalDateTime.now(),
         ex.getMessage(),
              request.getDescription(false));
           }
}
```

`request.getDescription(false)`:

– Whether to include client-specific information like session id and user name

application.yml

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test
    username: root
    password: srivatsan
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    show-sql: true
    hibernate:
       ddl-auto: update
       naming:
        physical-strategy:
    org.hibernate.boot.model.naming.PhysicalNamingStrategyStand
    ardImpl
```

application.yml

```
logging:
  level:
    org.hibernate.sql:  debug
    org.hibernate.type.descriptor.sql.BasicBinder: trace
```

QUERY DSL

Query DSL

- **Domain Specific Language**
  - Extension of a programming language to address a domain.
  - Enhancing Java to be better suited at creating and working with JPA queries.
  - A Powerful, flexible and customizable based on JPA entity
  - Framework checks the validity of queries when application starts up
  - Have been in use in scripted language

Query Method Syntax

- Methods are defined in JPA repository that Spring Data JPA will auto-implement
  - The query parser will look for methods that start with **find, query, read, count, or get.**

- Prefixes can be enhanced with other keywords until it gets to the B-Y, or By, of the method name.

- findByAgeLike(Integer age);

Query Method Syntax

- findByAgeLike(Integer age);
- find
  - Method starts with find so that the query parser understands that it needs to implement this query contract.
- By
  - Following the find keyword,
  - Signaling that the criteria information will be coming next in the method name.
- Age
  - Matches the attribute name age JPA entity, and age is of data type Integer.

Query Method Syntax

- findByAgeLike(Integer age);

- Like
  - final keyword tells the implementation to create a Like query
  - Need to pass variable that the query implementation should use as the actual filter criteria.
  - It's of type Integer because data type of age in entity is of type Integer.

Custom Repository Methods

- Can add custom methods to the interface
- **Methods are Derived by two main parts separated by the find *By* keyword**

- **Equality Condition Keywords**
  - **List** findByNameEquals(**String** name);
  - **List** findByNameIsNot(**String** name);

- **Similarity Condition Keywords**
  - **List** findByNameStartingWith(**String** prefix);
  - **List** findByNameContaining(**String** infix);

Custom Repository Methods

- **Comparison Condition Keywords**
  - List findByAgeLessThan(Integer age);
  - List findByAgeLessThanEqual(Integer age);
  - List findByAgeGreaterThan(Integer age);

- **Multiple Condition Expressions**
  - List findByNameOrBirthDate(String name, ZonedDateTime birthDate);
  - List findByNameOrBirthDateAndActive(String name, ZonedDateTime birthDate, Boolean active);

- **Sorting the Results**
  - **List** findByNameOrderByName(**String** name);
  - **List** findByNameOrderByNameAsc(**String** name);

Custom Repository Methods

- findByStartDateBetween
- findByStartDateAfter
- findByAgeOrderByLastnameDesc.

- ***To Return Just the two properties Name and email***

```
public interface ProjectNameAndEmail {

 String getName();
 String getEmail();
 }

List<ProjectNameAndEmail> findAllById(int id);
```

@Query

- Used to write the more flexible query to fetch data.
- Supports both JPQL and native SQL queries.
- By default will be using JPQL to execute the queries.
  - Using normal SQL queries, would get the query syntax error exceptions.
- Can use native SQL queries by setting nativeQuery flag to true.
  - Pagination and dynamic sorting for native queries are not supported in spring data jpa.

@Query

- Used to write the more flexible query to fetch data.

- Supports both JPQL and native SQL queries.

- By default will be using JPQL to execute the queries.
  - Using normal SQL queries, would get the query syntax error exceptions.

- Can use native SQL queries by setting nativeQuery flag to true.
  - Pagination and dynamic sorting for native queries are not supported in spring data jpa.

@Modifying

- @Query
  - Can also be used for queries that add, change, or remove records in database.
- @Modifying
  - Indicates a query method should be considered as modifying query as that changes the way it needs to be executed.
  - Not applied on custom implementation methods or queries derived from the method name
  - Those methods already have control over the underlying data access APIs or specify if they are modifying by their name.

@Modifying

- Method annotated with @Modifying also requires @Transactional
- By default, CRUD methods on repository instances are transactional.
  - For read operations, the transaction configuration readOnly flag is set to true.
  - For other operations are configured with a plain @Transactional
  - We need to add @Transactional annotation with required attribute values.

@Query - Example

```
@Repository
public interface TourGuideRepository extends
    JpaRepository<TouristGuide, Integer> {

public TouristGuide findByFirstName(String name);

 @Query(value = "SELECT * FROM tourguides WHERE
    first_name = :firstName", nativeQuery = true)

List<TouristGuide> findGuideByFirstName(@Param("firstName")
    String firstName);
```

@Query - Example

```
@Query("FROM TouristGuide WHERE firstName =
 :firstName AND lastName = :lastName")
 List<TouristGuide>
 findByFirstAndLastName(@Param("firstName") String
 firstName, @Param("lastName") String lastName);



@Query("UPDATE TouristGuide SET firstName = :prefix ||
 firstName")
@Modifying
@Transactional
void updatePrefix(@Param("prefix") String prefix);


}
```

Application.yml

```
server:
  port: 2525
spring:
  datasource:
    username: root
    password: srivatsan
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/test
  jpa:
    hibernate:
      ddl-auto: update
      naming:
        physical-strategy:
org.hibernate.boot.model.naming.PhysicalNamingStrategyStanda
rdImpl
    show-sql: true
```

•

DOCUMENTING REST API

Documentation

- Essential part of building REST APIs
- **Open Api Documentation**
  - *Standard, programming language-agnostic interface description for REST APIs,*
  - *Allows both humans and computers to discover and understand the capabilities of a service*
  - *No need to access source code,*
  - *Can add additional documentation, or inspect network traffic*

Spring Documentation

– A tool that simplifies the generation and maintenance of API docs

– Based on the OpenAPI 3 specification, for Spring Boot 1.x and 2.x applications.

– Required dependency are added to *pom.xml*:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.3.0</version>
</dependency>
```

View the Documentation

- Can be viewed after executing the application
- Its rendered in the browser in JSON format by default.
- **Default Path** :   http://localhost:8080/v3/api-docs/
- *Can be Customised using application.properties* file:

  springdoc.api-docs.path=/api-docs

- **Custom Path**   :   http://localhost:8080/api-docs/

Swagger

- springdoc-openapi is integrated with Swagger UI
- Access from the folowing URL
- http://localhost:8080/swagger-ui.html

• Can be customized using application.properties

springdoc.swagger-ui.path=/swagger-ui-custom.html

Custom Configuration

– Documenting at the Class Level

```
@SpringBootApplication
@OpenAPIDefinition( info = @Info(title = “Invoice Service")
)
```

– Can also Document at the Method Level

```
@Operation(description = "Method to Add Invoice")
```

Can also Document at the Model Class

– Using JSR 303 Annotations

Best practices

- Use Nouns for Resource Identification
- Use Proper Http Headers for Serialization formats
- Get Method and Query Parameter should not alter the state
- Always use plurals when you name resources.
- Use Appropriate Http Response Status Codes
- Field Name casing Conventions
- Provide Searching, Sorting and Pagination
- Use Versioning of API
- Provide Links for Navigating through API – Hateos
- Handle Error Properly
- Using DTO to Handle the Request and Response

REACTIVE SPRING

Introduction

- A programming paradigm that promotes
  - ***asynchronous, non-blocking, event-driven approach to data processing.***

- ***Thread per request model*** to **more requests with few number of threads**
  - Prevent threads from blocking while waiting for I/O operations to complete

Subscription

- Subscription made by the **subscriber**
- With the **publisher**
  - To fetch data.

```
public interface Subscription {
public void request(long n);
public void cancel();
}
```

Subscriber

- Represents the consumer of the stream data.

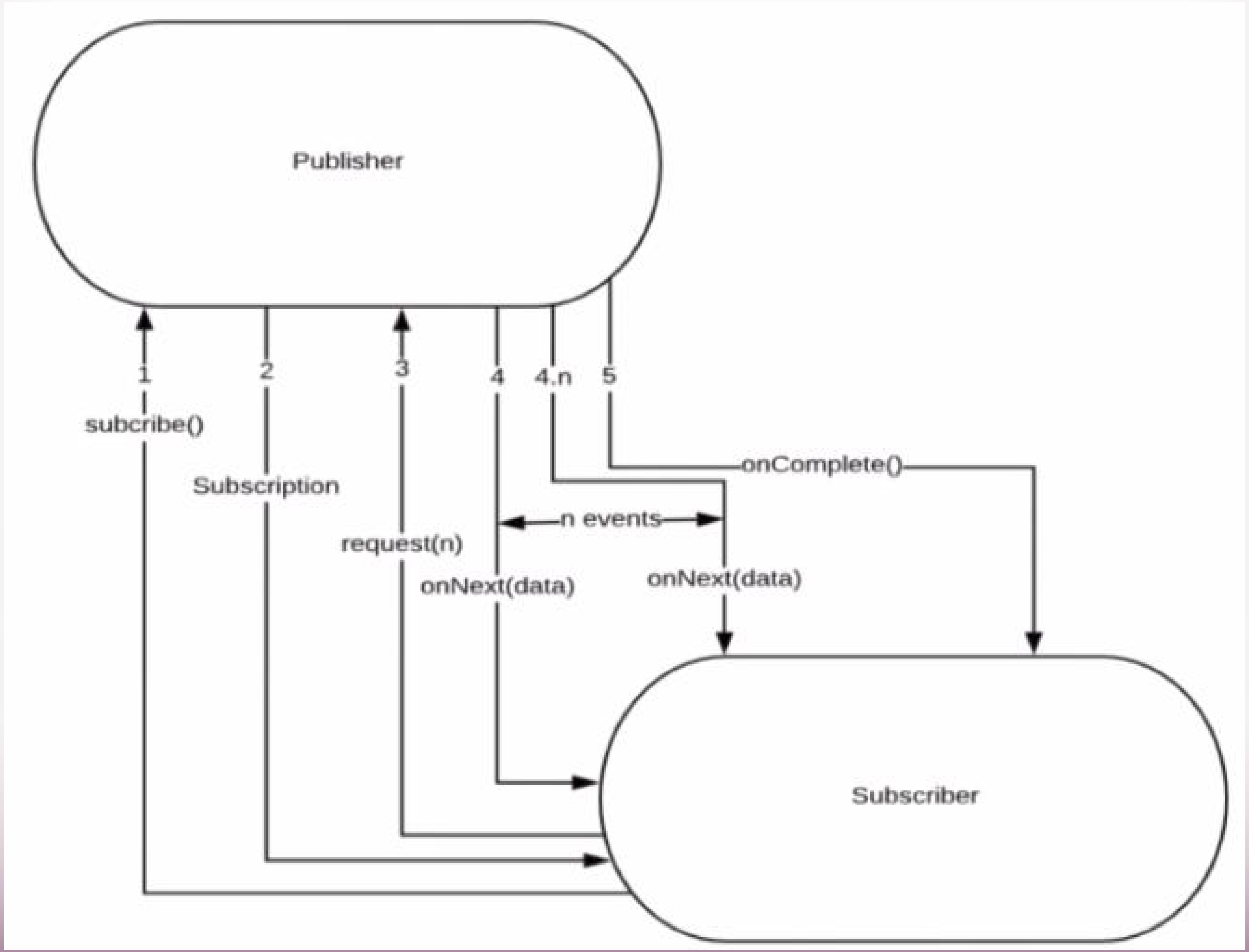
```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Publisher

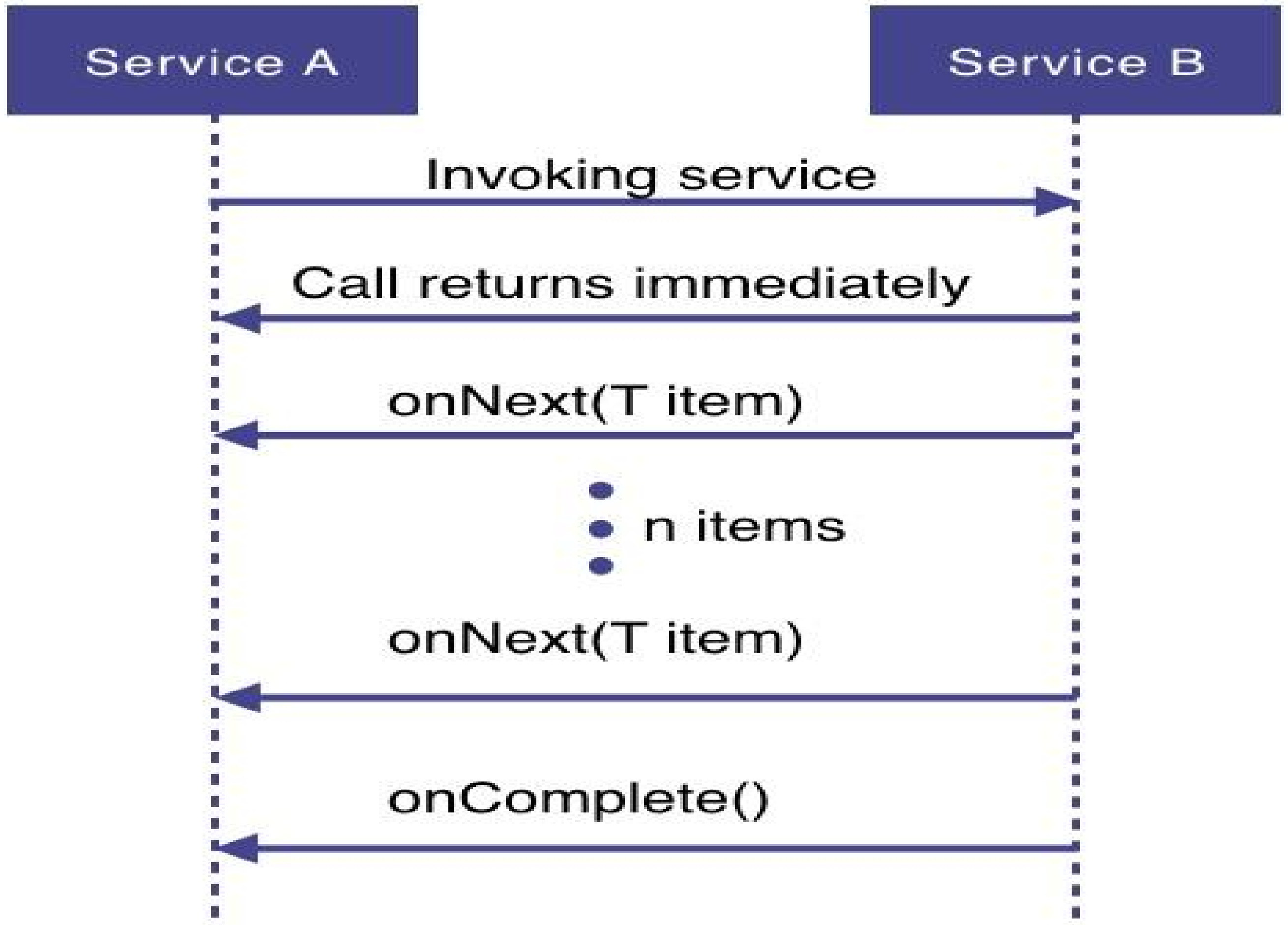
- Represents the data source
  - database, external service, etc.

```
public interface Publisher<T> {

public void subscribe(Subscriber<? super T> s);

}
```

Flow Between Publisher and Subscriber



How Does it Work



How Does it Work

1. Service A wants to retrieve some data from service B.

2. Service A will make a request to service B which returns immediately
   – Non-blocking and asynchronous.
3. Data requested will be made available to service A as a data stream,
   – Service B will publish an onNext-event for each data item one by one.
4. When all the data has been published
   – onComplete event is fired
5. In case of an error, an onError event would be published
   – Items would not be emitted.

PROJECT REACTOR

PROJECT REACTOR

- Spring Framework supports reactive programming since version 5.
  - That support is build on top of Project Reactor.

- WebFlux, Spring's reactive-stack web framework, requires Reactor as a core dependency.

- Suited for Microservices Architecture
- Offers backpressure-ready network engines for HTTP (including Websockets), TCP, and UDP.
- Built and maintained by Pivotal
- Recommended Library to work with Spring Boot
- Runs on Java 8 and above.

PROJECT REACTOR

- Spring Framework supports reactive programming since version 5.
  - That support is build on top of Project Reactor.
- WebFlux, Spring's reactive-stack web framework, requires Reactor as a core dependency.
- Suited for Microservices Architecture
- Offers backpressure-ready network engines for HTTP (including Websockets), TCP, and UDP.
- Built and maintained by Pivotal
- Recommended Library to work with Spring Boot
- Runs on Java 8 and above.

Setting up a Project

- Spring Boot application using Spring Initializr.
  - Add Spring Reactive Web as dependecy
- The spring-boot-starter-webflux dependency is added
  - It will bring in the reactor-core dependency.
  - The reactor-test has been added as a dependency.
- **Spring Webflux**
  - A **reactive**-stack web framework
  - Runs on Netty, and Servlet 3.1+ containers.

Types of Publishers

- Spring Webflux uses two Publishers
- **Mono**
  - To publish 0..1 element
  - Mono is like the Optional type provided in Java 8.
  - Mono to return a single object or VOid
- **Flux**
  - To publish 0..N element.
  - Used to return lists.

Flux and Mono

- They are both terminated either by a completion signal or an error
- They call a downstream Subscriber's onNext, onComplete and onError methods.
- Both provide a set of operators to support transformations, filtering, and error handling.

Creating Flux

- **Just (T... data)**
  - Overloaded method
  - Create a new Flux that will only emit a single element then onComplete.
  - Returns a new Flux

- Flux<String> fluxColors = Flux.just("red", "green", "blue");

- fluxColors.subscribe(System.out::println);

Mono

- Emits at most one item and then optionally terminates with an onComplete signal or an onError signal.

```
Mono noData = Mono.empty();

Mono data = Mono.just(“ramesh”);
```

Mono and Flux

```
public Mono<Book> bestBook(){
    Mono<Book> bestBook = Mono.just(bookList.get(1));
return bestBook;
}

public Mono<Book> betterBook(){
   Mono<Book> betterBook = Mono.just(bookList.get(0));
return betterBook;
}


       public Flux<Book> fluxFromMono(){

             return Flux.concat(bestBook(),betterBook());
        }
```

SPRING WEBCLIENT

WebClient

- Introduced from Spring 5
  - Its a non-blocking client with support for Reactive Streams.
  - It’s a replacement for *RestTemplate* .
  - Its more functional and is fully reactive.
  - It's included in the spring-boot-starter-weblux dependency

Web Client

```
 @Bean
WebClient.Builder builder() {
        return WebClient.builder();
    }

    @Bean
    WebClient client(WebClient.Builder builderRef) {
        return builderRef.build();
    }
```

Http Methods

- **get()**
  - indicates that we are making a *GET* request.
  - We know that the response will be a single object, so we're using a Mono as explained before.

  - `client.get()`
  - `Client.post()`

Rest Client - Controller

```
@RestController
public class ClientController {
@Autowired
private WebClient client;

@GetMapping(path = "/hotels")
public Flux<String> getAllHotels(){
return client.get()
   .uri("http://localhost:7075/api/v1/hotels")
    .retrieve()
     .bodyToFlux(String.class);
     }
 }
```

Rest Client - Controller

```
@PostMapping(path = "/hotels")
public Mono<HotelDto> create(@RequestBody HotelDto
dto)
{
    return client.post()
        .uri(" http://localhost:7075/api/v1/hotels")
        .body(Mono.just(dto), HotelDto.class)
        .retrieve()
        .bodyToMono(HotelDto.class);
}
```

Rest Client - Controller

```
@DeleteMapping(path = "/hotels/{id}")
public Mono<Void> removeHotelById(@PathVariable("id")
int id){


  return client.delete()
      .uri(" http://localhost:7075/api/v1/hotels/{id}"
,id)
         .retrieve()
         .bodyToMono(Void.class);
}
}
```

Rest Client - Controller

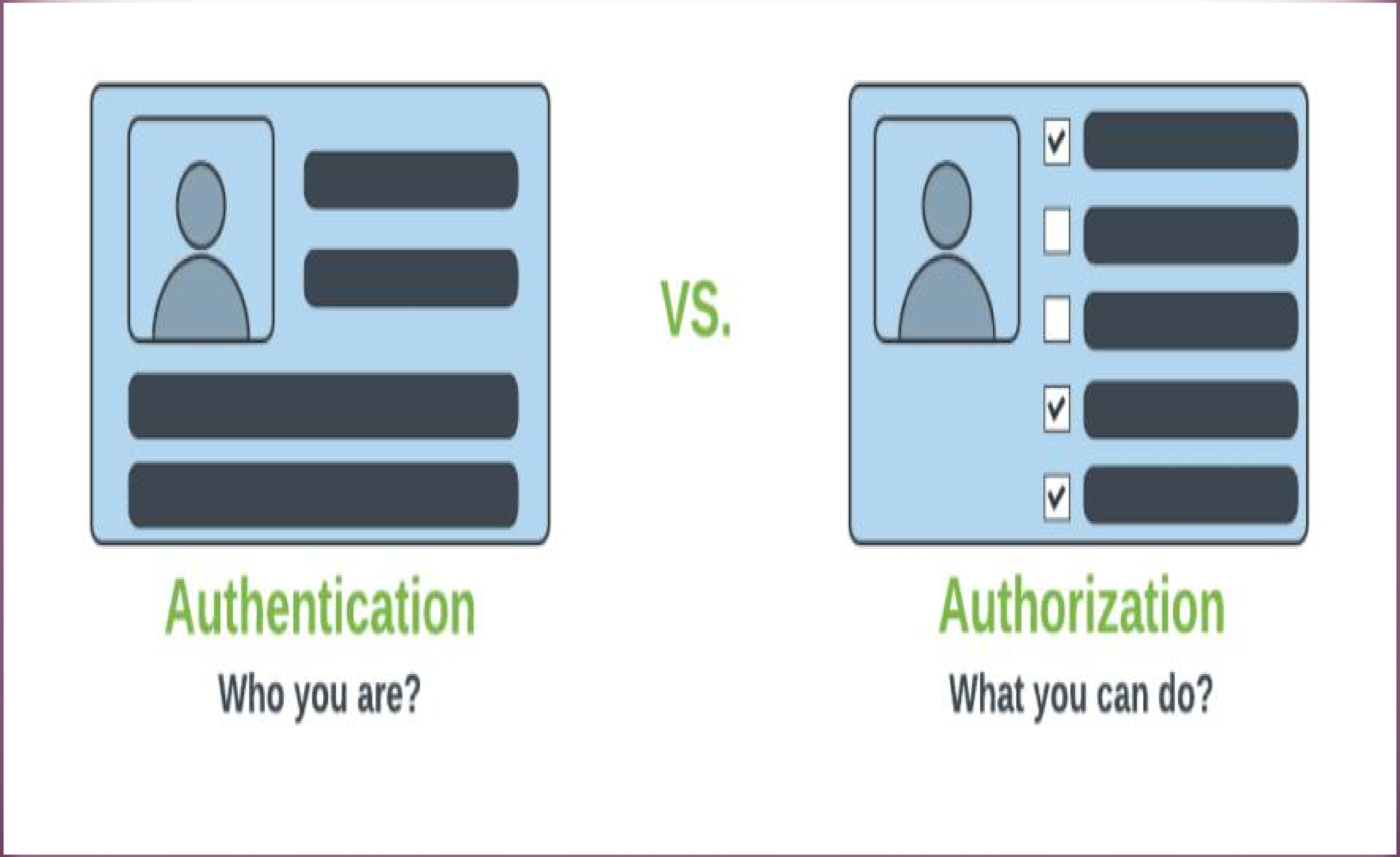
```
@GetMapping(path = "/hotels/{id}")
public Mono<String> getHotelById(@PathVariable("id")
int id){


return client.get()
   .uri(" http://localhost:7075/api/v1/hotels
/{id}",id)
    .retrieve()
     .bodyToMono(String.class);


}
```

SPRING SECURITY

Spring Security

- Highly customizable authentication and access-control framework.
- Focuses on providing both authentication and authorization
- “**Authentication**”
  - The process of establishing a user, device or some other system
- “**Authorization**”
  - The process of deciding whether a principal is allowed to perform an action within the application.
- *Authorization is done after authentication process.*
  - Authenticated user may have one or more “roles”

Authentication & Authorization



@EnableWebSecurity

- **Can be Added to** the Class annotated with @Configuration or @SpringApplication
  - Create a Class of   type **WebSecurityConfigurerAdapter**
  - Spring finds bean with @Configuration and automatically apply Security

Spring Security Modules

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

WebSecurityConfigurerAdapter

- A base class for creating a WebSecurityConfigurer instance.
- Can do customization by overriding methods
- **configure(HttpSecurity)**
  - Defines the mapping of secured URLs or path that will determine if the user can access specific pages.
- **configure(AuthenticationManagerBuilder)**.
  - Defines how the security will handle the retrieval of user information commonly in the database.

configure(HttpSecurity)

- **HttpSecurity**
  - Provides a Method authoriseRequests()

- **authorizeRequests()**
  - To Define custom requirements for URLs
  - Can add multiple children to this method.
  - The matchers are considered in the order they were declared.

```
http.authorizeRequests()
  .anyRequest().authenticated()
  .and().httpBasic();
```

configure(HttpSecurity)

- **public HttpSecurity antMatcher(java.lang.String antPattern)**
  - Used to configure to be invoked when matching the provided ant pattern.
- **antPattern**
  - the Ant Pattern to match on (i.e. "/admin/**")
- **formLogin()**
  - Specifies to support form-based authentication.
  - If loginPage(String) is not specified a default login page will be generated.
- **httpBasic()**
  - Configures HTTP Basic authentication.

configure(AuthenticationManagerBuilder).

- **AuthenticationManagerBuilder**
  - Used to create an Authentication Manager
- **Authentication Manager**
  - Used to authenticate the passed **Authentication** object,
  - Allows building in memory authentication
  - Can also use JDBC based authentication,

Form Login

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests().antMatchers("/first").authenticated().and().formLogin();

}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws
    Exception {

auth.inMemoryAuthentication().withUser("ramesh").password("{noop}ramesh").roles
    ("USER");
}
```

Controller

```
@RestController
public class ProjectController {

@GetMapping("/first")
public String getSecuredMessage(Principal user) {

return "Welcome to Secured Page"+user.getName();
}

@GetMapping("/second")

  public String getUnSecuredMessage() {

return "Welcome to UN-Secured Page";
}
}
```

Sample Security Config

```
http.authorizeRequests()
.antMatchers("/","/*.html","/h2/**")
 .permitAll().
 antMatchers("/first")
                    .authenticated().and()
                     .httpBasic().and()
                     .csrf().
                    disable();
```

Method Level Security

- Spring Security supports authorization semantics at the method level.
- Service layer can be secured by which roles are able to execute a particular method – and test it using dedicated method-level security test support.

Method Level Security

```
@Configuration
@EnableGlobalMethodSecurity(jsr250Enabled =  true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {


protected void configure(HttpSecurity http) throws Exception {
   // required configurations
 }
 protected void configure(AuthenticationManagerBuilder auth) throws
    Exception {
          // required configurations
     }
 }
```

Method Security

- **@RolesAllowed**
  - **is** a standard annotation

```
@PermitAll
 public String anonymously() {
    return "Hello, World!";  }


@RolesAllowed({"ROLE_ADMIN"})
public String hasRole() {
return "Hello, World!";
}
```

Basic Authentication With WebClient

```
@Autowired
WebClient client;

@GetMapping(path = "/client")
public Flux<String> getOrders(){

return client.get().uri("http://localhost:8080/orders/pending")
        .headers(headers ->
headers.setBasicAuth("chennai","chennai"))
        .retrieve()
        .onStatus(HttpStatus::is5xxServerError, clientResponse ->
        Mono.error(new RuntimeException()))

        .bodyToFlux(String.class);
}
```