Spring Boot Introduction

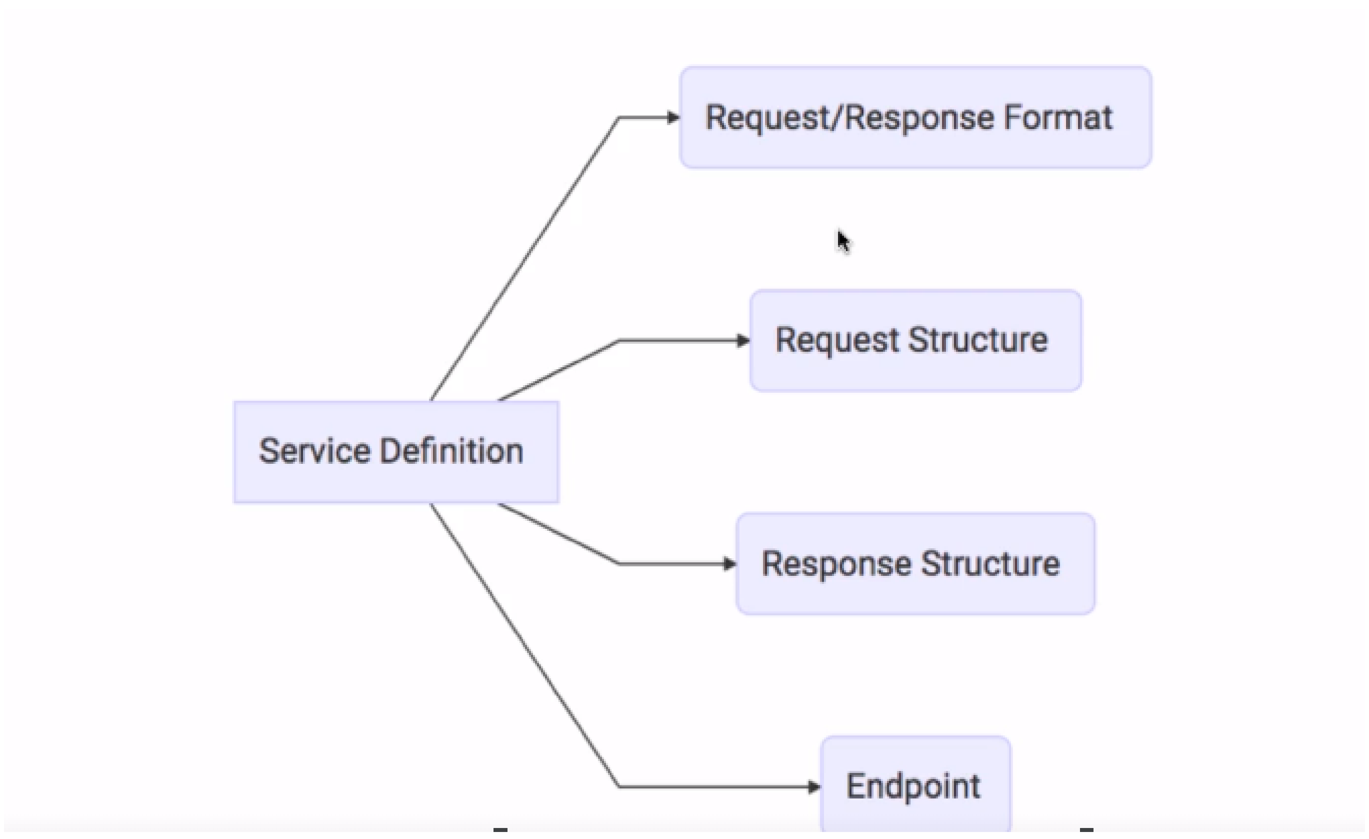
Follows front end controller pattern (**DispatcherServlet**) configured to root url by default.

All requests lands on dispatcher servlet and it looks at path and maps it to right controller.

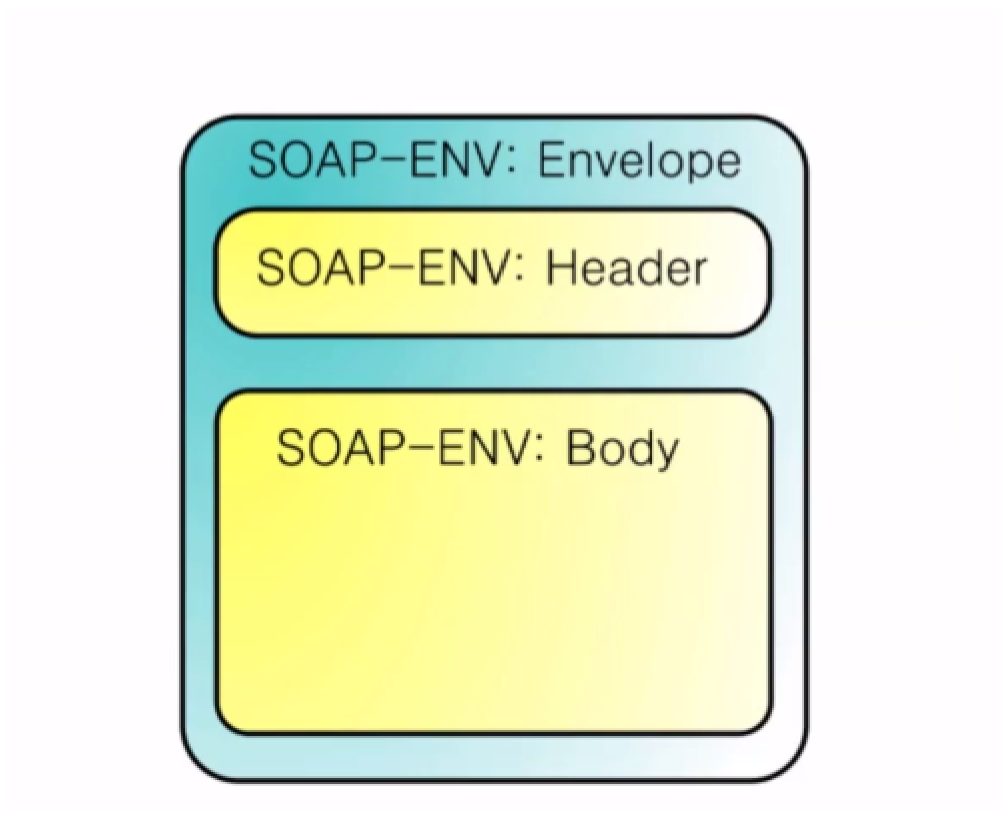
DispatcherServletAutoConfiguration is responsible for configuring the dispatcher servlet.

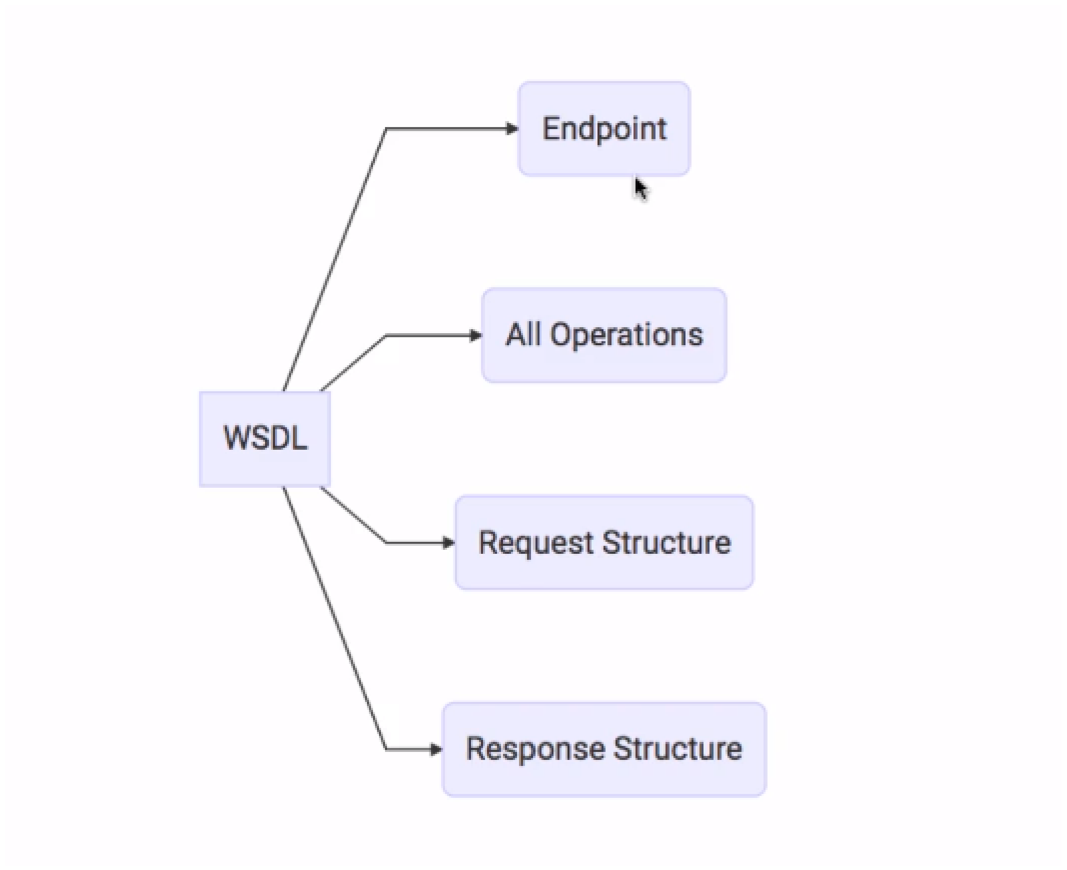
Java Objects are converted to JSON by @ResponseBody + JacksonHttpMessageConverters using **JacksonHttpMessageConvertersConfiguration**

Service Definition


Service Definition
Request/Response Format
Request Structure
Response Structure
Endpoint

Soap Request





REST API

Annotations

- Request Types:
    - @GetMapping
    - @PostMapping
    - @DeleteMapping
- Request Parameters:
    - @PathVariable
    - @RequestBody

Response Codes

- 200-success
- 201-created
- 204-no content
- 400-bad request
- 401-unauthorized
- 404-not found
- 500-internal server error.

Error Handling

ErrorMvcAutoConfiguration is responsible for mapping errors. If we need to override default we can extend **ResponseEntityExceptionHandler**

```
import java.time.LocalDateTime;

import org.apache.commons.lang3.exception.ExceptionUtils;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandl
er;
import lombok.AllArgsConstructor;
import lombok.Getter;

/** Custom Response Entity Exception Handler*/
@ControllerAdvice
public class CustomizedResponseEntityExceptionHandler extends
ResponseEntityExceptionHandler{

    @ExceptionHandler(Exception.class)
    public final ResponseEntity<Object> handleAllException(Exception ex,
WebRequest request) throws Exception {
        HttpStatus httpStatus=null;
        if(ex instanceof UserNotFoundException) {
            httpStatus = HttpStatus.NOT_FOUND;
        }
        ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
                ex.getMessage(),
                request.getDescription(false),
                ExceptionUtils.getStackTrace(ex));
        return new ResponseEntity<>(errorDetails,null!=httpStatus?
httpStatus:HttpStatus.INTERNAL_SERVER_ERROR);
    }

}

@AllArgsConstructor
@Getter
class ErrorDetails {
    private LocalDateTime timestamp;
    private String message;
    private String requestDetails;
    private String stackTrace;
}
```

Request Validations

- U can achieve request validation using spring-boot-starter-validation dependency in pom.xml
- In model class add required validations.

```java
package com.kaivikki.model;
import java.time.LocalDate;
import jakarta.validation.constraints.Past;
import jakarta.validation.constraints.Size;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class User{

    private Integer id;

    @Size(min = 2, message = "name should have atleast two characters")
    private String name;

    @Past(message = "birth date should be in past")
    private LocalDate birthDate;
}
```

- In Controller add @Valid

```
@PostMapping(path = "/users")
    public ResponseEntity<User> saveUser(@Valid @RequestBody User user) {
        return new ResponseEntity<User>(userDao.save(user), HttpStatus.CREATED);
    }
```

- In Custom exception handler override handleMethodArgumentNotValid method

```
protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
HttpStatusCode status, WebRequest request) {
        List<FieldError> fieldErrors = ex.getFieldErrors();
        JsonObject jsonObject = new JsonObject();
        fieldErrors.forEach(fe->{
            jsonObject.addProperty(fe.getField(), fe.getDefaultMessage());
        });

        ErrorDetails errorDetails = new ErrorDetails(LocalDateTime.now(),
                jsonObject.toString(),
                request.getDescription(false),
                ExceptionUtils.getStackTrace(ex));
        return new ResponseEntity<>(errorDetails,HttpStatus.BAD_REQUEST);
    }
```

REST API Documentation.

- **OpenAPI Specification**: (Earlier called as Swagger Specification)
    - Provide Documentation to discover and understand rest api.
    - You can either write this documentation manually
    - Or, You can automate the api documentation using libraries.
    - springdoc-openapi is one such java library which helps to automate the generation of API documentation for spring boot project.

```xml
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

- **Swagger UI**
    - Visualize api documentation and interact with your apis

Content Negotiation:

Consumer can tell the REST API provider what they want by provinding headers like:

- Accept Header : For response format
- Accept-Language Header : For response language.

```
// For Xml format we use the dependency.
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>

// For messages in diffrent language use MessageResource which will read
messages_<Accept_Language_Header_Value>.properties from classpath.
@Autowired
MessageSource messageSource;

@GetMapping(path = "/greetings")
public String greet() {
    return messageSource.getMessage(
    "good.morning.message",
    null,
    "Namaste",
    LocaleContextHolder.getLocale());
}
```

Version REST API

Versioning of REST API can be acheived via one of the below listed channels:

- **Versioning via URI**: This will create different URL for each version.
- **Versioning via Request Param.**: This will create different URL for each version.
- **Versioning via Request Header.**: In this URL remains the same, but problem is incorrect usage of headers

```java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import com.kaivikki.model.Name;
import com.kaivikki.model.PersonV1;
import com.kaivikki.model.PersonV2;

@RestController
public class PersonContoller {

    // http://localhost:8080/api/v1/person
    @GetMapping("/api/v1/person")
    public PersonV1 getPerson() {
        return new PersonV1("Vikram Arora");
    }

    // http://localhost:8080/api/v2/person
    @GetMapping("/api/v2/person")
    public PersonV2 getPersonV2() {
        Name name = new Name();
        name.setFirstName("Vikram");
        name.setLastName("Arora");
        return new PersonV2(name);
    }

    // http://localhost:8080/person?version=1
    @GetMapping(path = "person", params = "version=1")
    public PersonV1 getPersonWithVersionInRequestParam() {
        return getPerson();
    }

    // http://localhost:8080/person?version=2
    @GetMapping(path = "person", params = "version=2")
    public PersonV2 getPersonV2WithVersionInRequestParam() {
        return getPersonV2();
    }

    // http://localhost:8080/person
    @GetMapping(path = "person", headers = "version=1")
    public PersonV1 getPersonWithVersionInRequestHeader() {
        return getPerson();
    }
    // http://localhost:8080/person
    @GetMapping(path = "/person", headers = "version=2")
    public PersonV2 getPersonV2WithVersionInHeader() {
        return getPersonV2();
    }
}
```

HATEOAS for REST API's

Hypermedia as engine of application stage. We have introduce actions in our resp api responses as links using hateoas.

```
// Add a dependency for spring hateoas.
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>

// Rest API method should now return EntityModel with links.
// Example below get user by id response has link to get all users api
@GetMapping(path = "/users/{id}")
public EntityModel<User> getUserById(@PathVariable Integer id) {
    User user = userDao.findOne(id);
    if (null == user) {
        throw new UserNotFoundException("No User Found With Id " + id);
    }
    EntityModel<User> userEntityModel = EntityModel.of(user);
    WebMvcLinkBuilder webMvcLinkBuilder =
WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder
    .methodOn(this.getClass())
    .getAllUsers()
    );
    userEntityModel.add(webMvcLinkBuilder.withRel("all-users"));
    return userEntityModel;
}

@GetMapping(path = "/users")
public List<User> getAllUsers() {
    return userDao.findAll();
}

//Api Response will look like below:
{
    "id": 1,
    "name": "Vikram",
    "birthDate": "1993-04-14",
    "_links": {
        "all-users": {
            "href": "http://localhost:8080/users"
        }
    }
}
```

REST API Static Filtering

- **Serialization**: Converting objects to streams (example JSON). Most popular JSON serialization in Java is via Jackson.
- We can customize the REST API response returned by Jackson framework.
    - **@JsonProperty**: Customize the field names in response.
    - **@JsonIgnore**: Static Filter the property in the response. This is applied at the property level.
    - **@JsonIgnoreProperties**: Static filter properties in the response. This is applied at the class level.

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Getter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class FilterBean {

    @JsonProperty("field_1")
    private String field1;

    @JsonIgnore
    private String field2;

    private String field3;

}
```

REST API Dynamic Filtering

- Return different attribute of the same bean in differnt api's.
- We can achieve this by passing serializing instructions to the Jackson coverter using MappingJacksonValue.
    - **@JsonFilter** : Applied at class level specify the filter name.

```
import com.fasterxml.jackson.annotation.JsonFilter;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.ToString;

@Getter
@NoArgsConstructor
@AllArgsConstructor
@ToString
@JsonFilter("FilterBeanField2Filter")
public class FilterBean {
    private String field1;
    private String field2;
    private String field3;
}

//Api Handler with dynamic filtering
//Dynamic Filter Example to filter field 2 from API response.
    @GetMapping("/testFilterField2")
    public MappingJacksonValue filterField2() {
        FilterBean filterBean = new FilterBean("value1", "value2", "value3");
        MappingJacksonValue mappingJacksonValue = new
MappingJacksonValue(filterBean);
        SimpleBeanPropertyFilter filter =
SimpleBeanPropertyFilter.filterOutAllExcept("field1","field3");
        FilterProvider filters = new
SimpleFilterProvider().addFilter("FilterBeanField2Filter", filter );
        mappingJacksonValue.setFilters(filters);
        return mappingJacksonValue;
    }
```

Spring Boot Actuator

- Spring boot actuator provides spring boot production ready features.
- Monitor and manage your application in production.
- Add a dependency for spring boot starter actuator.
- Provides number of endpoints:
  - **beans** : Complete list of spring beans in your app.
  - **health**: Application health information.
  - **metrics**: Application metrics
  - **mappings**: Details around Request Mappings
  - And a lot more.
- By default actuator only exposes application health end points. If you need to expose other end points via actuator add a property called: **management.endpoints.web.exposure.include=***

```
//<!-- Actuator dependency in pom.xml -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

//Actuator API is availble at path /actuator
//http://localhost:8080/actuator
```

Spring Data JPA

Using in memory H2 Database.

- Add a dependency in pom.xml for h2 database.
- Enable console by adding a property spring.h2.console.enabled=true

```
// H2 Database dependency
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

```
#        Properties to add in application.properties
# Property to enable H2 console.
spring.h2.console.enabled=true
# Property to define which db to create
spring.datasource.url=jdbc:h2:mem:testdb
# Property to defer data source initialation(data load from data.sql) till db is
created.
spring.jpa.defer-datasource-initialization=true
```

Using MySQL Database

```
#### MySQL Database SETUP Via Command Line ###########.

#Show all databases
show databases;

#Show all users;
select user, host from mysql.user;


# create database spring_boot_all_in_one_db
mysql> create database springdb;
# create user springuser
mysql> create user 'springuser'@'localhost' identified by 'Password';
# grant previlage to springuser on spring_boot_all_in_one_db
mysql> grant all on springdb.* to 'springuser'@'localhost';

#Verify
SHOW GRANTS FOR 'springuser'@'localhost';

########## Application.properties#########
#Spring MYSQL Datasource
spring.datasource.url=jdbc:mysql://localhost:3306/springdb
spring.datasource.username=springuser
spring.datasource.password=Password

#JPA Properties.
# tell spring to create tables for entities on server startup
spring.jpa.hibernate.ddl-auto=update
# show hiberate sqls in logs
spring.jpa.show-sql=true
# dialect to user
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

```
// Add mysql dependency in pom.xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Spring Security