Berner
Fachhochschule

Spring REST Controller

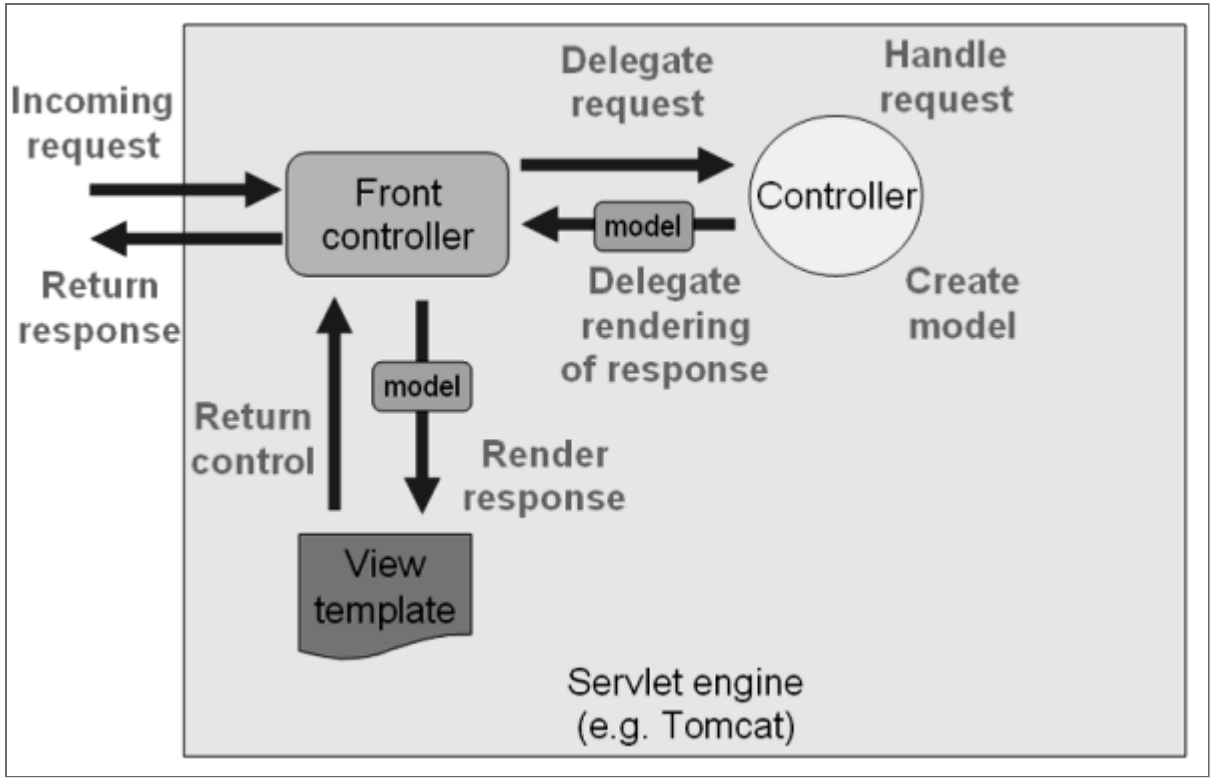
Stephan Fischli

Introduction

- Spring Web MVC is the original web framework built on the Java Servlet specification
- Spring Web MVC supports the development of web applications with server-side HTML rendering and RESTful web services
- Spring Boot can be used to create stand-alone web applications (with an embedded servlet container) or traditional WAR deployments
- Spring 5.0 introduced a reactive-stack web framework called Spring WebFlux

Dispatcher Servlet

- When using the starter `spring-boot-starter-webmvc`, the servlet `DispatcherServlet` is automatically registered with the servlet container
- This servlet implements the **front controller pattern**, i.e. it is only responsible for the request dispatching, the actual processing is performed by configurable delegate components


Incoming request
Delegate request
Handle request
Front controller
Controller
model
Return response
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)

Web Application Context

- The `DispatcherServlet` uses a `WebApplicationContext` which contains the following beans among others
  - `HandlerMapping` maps requests to handlers
  - `HandlerAdapter` helps to invoke handlers, e.g. based on annotations
  - `HandlerExceptionResolver` resolves exceptions to handlers, HTML error views, or other targets
  - `ViewResolver` resolves logical view names returned from a handler to an actual view to render the response
- Furthermore, additional features are auto-configured such as message converters

REST Controllers

Introduction

- Spring MVC provides an annotation-based programming model for RESTful web services
- REST Controllers are components annotated with `@RestController`
- Other annotations are used to define request mappings, request input, exception handling, and more

```java
@RestController
public class HelloController {

    @GetMapping
    public String sayHello() {
        return "Hello World!";
    }
}
```

Request Mapping

- The `@RequestMapping` annotation is used to map requests to controller methods based on the request path
- At the class level it defines a shared mapping, at the method level it narrows the mapping to a specific endpoint
- `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping` and `@DeleteMapping` are HTTP method specific shortcut variants

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @GetMapping("/count")    // @RequestMapping(path = "/count",
method = GET)
    public int getCustomerCount() {
        ...
    }
    @PostMapping    // @RequestMapping(method = POST)
    public void registerCustomer(@RequestBody Customer customer) {
        ...
```

```
        }
}
```

Path Variables

- Requests can be mapped using path variables in curly brackets
- The corresponding values can be accessed through method parameters annotated with `@PathVariable`
- The method parameter has to be named explicitly, if its name does not match the name of the path variable
- Path variables are automatically converted to the parameter type

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @GetMapping("/{id}")
    public Customer getCustomer(@PathVariable long id) {
        ...
    }
    @GetMapping("/{id}/orders/{orderId}")
    public Order getOrder(
        @PathVariable("id") long customerId, @PathVariable long
orderId) {
        ...
```

```
        }
}
```

Request Parameters

- Request parameters can be accessed using the `@RequestParam` annotation
- Request parameters are submitted either as query parameters of the request URI or as form parameters in the request body
- By default, request parameters are required but they can be defined as optional using the `required` attribute

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @GetMapping("/{id}/orders")
    public List<Order> getOrders(@PathVariable long id,
        @RequestParam(required = false) int year) {
        ...
    }
}
```

Request Parameters (cont.)

- The request mapping can be narrowed based on the presence, absence or value of request parameters using the `params` attribute

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @GetMapping(path = "/{id}/orders", params = "year")
    public List<Order> getOrders(@PathVariable long id, @RequestParam
int year) {
        ...
    }
    @GetMapping(path = "/{id}/orders", params = "!year")
    public List<Order> getOrders(@PathVariable long id) {
        ...
    }
}
```

Header Parameters

- A request header can be bound to a method argument using the `@RequestHeader` annotation with its name
- As with request parameters, header parameters are required by default

```
@RestController
public class HelloController {

    @GetMapping
    public String sayHello(@RequestHeader("Accept-Language") String
language,
        @RequestHeader(value = "User-Agent", required = false) String
agent) {
        ...
    }
}
```

Media Types

- The request mapping can be narrowed based on the `Accept` or `Content-Type` request header by specifying the media types that a controller method produces or consumes
- The default behavior is determined by the registered message converters

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @PostMapping(consumes = "application/json")
    public void registerCustomer(@RequestBody Customer customer) {
        ...
    }
    @GetMapping(path = "/{id}", produces = "application/json")
    public Customer getCustomer(@PathVariable long id) {
        ...
    }
    @GetMapping(path = "/{id}", produces = "text/plain")
    public String getCustomerAsString(@PathVariable long id) {
        ...
```

```
        }
}
```

HEAD and OPTIONS Requests

- Methods annotated with `@GetMapping` are implicitly mapped to support HEAD requests
- Instead of writing the response body, only the number of bytes are returned in the `Content-Length` response header
- OPTIONS requests are handled by setting the `Allow` response header to the HTTP methods of the controller methods that have matching URL patterns

Request and Response Body

- The `@RequestBody` annotation can be used to deserialize the request body into a parameter object using message converters
- The `@RestController` annotation contains the `@ResponseBody` annotation which is used to serialize the return value into the response body

```
@RestController
@RequestMapping("/news")
public class NewsController {

    @PostMapping
    public void postNews(@RequestBody NewsItem item) {
        ...
    }
    @GetMapping
    @ResponseBody    // not necessary
    public List<NewsItem> getNews() {
        ...
```

```
        }
}
```

Request and Response Entity

- A method parameter of type `HttpEntity` provides a container object that exposes request headers and body
- A return value of type `ResponseEntity` allows the response status and headers to be set

```
@RestController
@RequestMapping("/news")
public class NewsController {

    @GetMapping
    public ResponseEntity<List<NewsItem>>
getNews(RequestEntity<NewsItem> request) {
        String etag = request.getHeaders().getFirst(IF_NONE_MATCH);
        String currentTag = ...
        if (etag.equals(currentTag))
            return ResponseEntity.status(NOT_MODIFIED).build();
        else return ResponseEntity.status(OK).header(ETAG,
currentTag).body(news);
```

```
        }
}
```

Response Status

- A controller method can be annotated with `@ResponseStatus` to specify the HTTP response status if it returns successfully

```
@RestController
@RequestMapping("/news")
public class NewsController {

    @PostMapping
    @ResponseStatus(CREATED)
    public void postNews(@RequestBody NewsItem item) {
        ...
    }
}
```

Error Handling

Introduction

- If an exception occurs during request mapping or processing, it is delegated to a chain of `HandlerExceptionResolver` beans which generate an HTTP error response
- There exist built-in resolvers for Spring MVC exceptions, for custom exceptions, and for support of exception handler methods

Spring Exceptions

- The `DefaultHandlerExceptionResolver` is used to resolve Spring exceptions to their corresponding HTTP status codes, e.g.

Exception	Status Code
BindException	400 (Bad Request)
TypeMismatchException	400 (Bad Request)
HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)

Problem Details

- By default, Spring returns a proprietary JSON response when an error occurs
- The IETF standard **RFC 7807 Problem Details** can be enabled by setting the application property `spring.mvc.problemdetails.enabled`
- The standard specifies a generalized error schema:
  - `type` – an optional URI that identifies the error
  - `title` – a brief message about the error
  - `status` – the HTTP status code
  - `detail` – a human-readable explanation of the error
  - `instance` – a URI that identifies the occurrence of the error

```
{
    "type": "http://world.io/errors/offending-content",
    "title": "Bad Request",
    "status": 400,
    "detail": "News contains offending content",
```

```
    "instance": "/news"
}
```

Error Response Exception

- If an error occurs in a controller method, a `StatusResponseException` or an `ErrorResponseException` can be thrown
- The HTTP error response with the specified status is created by the `BasicErrorController` bean

```
@RestController
@RequestMapping("/news")
public class NewsController {

    @PostMapping
    public void postNews(@RequestBody News news) {
        if (...) {
            ProblemDetail detail = ProblemDetail.forStatusAndDetail(
                HttpStatus.BAD_REQUEST, "News contains offending
content");
            throw new ErrorResponseException(HttpStatus.BAD_REQUEST,
detail, null);
        }
        ...
```

```
        }
}
```

Custom Exceptions

- Custom exceptions can be annotated with `@ResponseStatus` or be derived from `ErrorResponseException` passing the error status and problem detail

```
@RestController
@RequestMapping("/news")
public class NewsController {

    @PostMapping
    public void postNews(@RequestBody News news) throws
InvalidNewsException {
        if (...) throw new InvalidNewsException("News contains
offending content");
        ...
    }
}
```

```
public class InvalidNewsException extends ErrorResponseException {
    public InvalidNewsException(String message) {
        super(HttpStatus.BAD_REQUEST,
            ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST,
message),
```

```
                null);
        }
    }
```

Exception Handler Methods

- Controllers may provide methods annotated with `@ExceptionHandler` to handle exceptions thrown by their methods
- The exception handler has full control of the error response to be returned to the client

```
@RestController
@RequestMapping("/news")
public class NewsController {
    private List<News> news = new ArrayList<>();

    @PostMapping
    public void postNews(@RequestBody News news) throws
InvalidNewsException {
        if (...) throw new InvalidNewsException("News contains illegal
characters");
        this.news.add(news);
    }

    @ExceptionHandler
    public ProblemDetail handle(InvalidNewsException ex) {
        return ProblemDetail.forStatusAndDetail(
            HttpStatus.BAD_REQUEST, ex.getMessage());
```

```
        }
}
```

Controller Advice

- In order to apply exception handlers across all controllers, they can be declared in a class annotated with `@RestControllerAdvice`

```
@RestControllerAdvice
public class RestControllerExceptionHandler {

    @ExceptionHandler
    public ProblemDetail handle(InvalidNewsException ex) {
        ProblemDetail detail =
ProblemDetail.forStatus(HttpStatus.BAD_REQUEST);
        detail.setTitle("Invalid news");
        detail.setDetail(ex.getMessage());
        detail.setType(...);
        return detail;
    }
}
```

REST Clients

Introduction

The Spring framework provides the following options for making REST calls:

- **RestTemplate** - synchronous client with template method API
- **RestClient** - synchronous client with a fluent API
- **WebClient** - non-blocking, reactive client with fluent API

RestClient

- The `RestClient` provides an abstraction of the HTTP protocol that enables convenient conversion between Java objects and HTTP message bodies
- An instance of the `RestClient` is created using one of its static `create` methods

```
RestClient restClient = RestClient.create("http://world.io");
```

- Alternatively, a builder can be used to configure a `RestClient`, e.g. to set a base URL, to specifiy default headers, or to add message converters

```
RestClient restClient = RestClient.builder()
    .baseUrl("http://world.io")
    .defaultHeader("Accept", "application/json")
    .messageConverters(converters -> converters.add(converter))
    .build();
```

Making a Request

- The HTTP request method can be specified with the `method` method or one of the convenience methods `get`, `post`, `put` etc.
- Request headers can be added with the `header` method or one of the convenience methods `accept`, `contentType`, `contentLength` etc.
- The request body can be set with the `body` method which internally uses HTTP message conversion

```
restClient.post()
    .uri("/news")
    .contentType(APPLICATION_JSON)
    .body(newsItem)
    .retrieve();
```

Retrieving the Response

- The HTTP response can be accessed by invoking the `retrieve` method
- The `body` method reads the content of the response body and converts it into the specified type

```
NewsItem news = restClient.get()
    .uri("/news/{id}", 172)
    .accept(APPLICATION_JSON)
    .retrieve()
    .body(NewsItem.class);
```

Accessing Response Details

- The method `toEntity` returns the response as a `ResponseEntity` object which provides access to the status code and headers of the response

```
ResponseEntity<NewsItem> response = restClient.get()
    .uri("/news/{id}", 172)
    .accept(APPLICATION_JSON)
    .retrieve()
    .toEntity(NewsItem.class);

System.out.println("Status: " + response.getStatusCode());
System.out.println("Headers: " + response.getHeaders());
System.out.println("Body: " + response.getBody());
```

Error Handling

- `RestClient` throws a subclass of `RestClientException` when a response with an error status code is retrieved
- Alternatively, an error handler can be registered using the `onStatus` method

```
NewsItem news = restClient.get()
    .uri("/news/{id}", 172)
    .accept(APPLICATION_JSON)
    .retrieve()
    .onStatus(HttpStatusCode::is4xxClientError, (request, response) ->
{
        HttpStatusCode status = response.getStatus();
        ...
    })
    .body(NewsItem.class);
```

REST Testing

Testing Principles

- Testing a RESTful web service is primarily about functional testing, i.e. ensuring that the service works correctly
- Tests can be divided into different categories
  - positive testing w/o optional parameter (happy paths)
  - negative testing with valid and invalid input
  - security testing (authentication/authorization)
  - destructive testing (robustness)
- For each test, various verifications must be made
  - correct HTTP status code, response headers and payload
  - correct application state
  - basic performance sanity
- There are also different test flows, e.g. isolated requests or workflow with multiple requests

Spring Boot Testing

- The `@SpringBootTest` annotation looks for a main configuration class and uses it to start the application context
- The `webEnvironment` attribute can be used to define how the tests are run
  - `MOCK` (default) provides a mock web environment without starting an embedded server
  - `RANDOM_PORT` provides a real web environment with a server on a random port
  - `DEFINED_PORT` provides a real web environment with a server on a defined port
  - `NONE` does not provide any web environment

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloTests {
    ...
}
```

Testing with Mock Environment

- By default, `@SpringBootTest` does not start an embedded server
- If web endpoints (including their request mapping) shall be tested, the **MockMvc** bean can be used to send requests to the controllers
- The requests are performed directly by invoking the `DispatcherServlet`
- This scenario allows testing an application's server side without the cost of starting a server

```
@SpringBootTest
@AutoConfigureMockMvc
public class HelloTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testHello() throws Exception {
        mockMvc.perform(get("/hello"))                          //
request builder
            .andExpect(status().isOk())                         // result
matcher
            .andExpect(content().contentType("text/plain"))
```

```
                .andExpect(content().string("Hello World!"));
    }
}
```

Testing with Mock Environment (cont.)

- Using the `@WebMvcTest` annotation, Spring Boot instantiates only the web layer rather than the whole application context
- If the tested controllers have dependencies, they must be mocked using the `@MockitoBean` annotation

```
@WebMvcTest
@AutoConfigureMockMvc
public class HelloTests {

    @Autowired
    private MockMvc mockMvc;
    @MockitoBean
    private GreetingService service;

    @Test
    public void testHello() throws Exception {
        Mockito.when(service.getGreeting()).thenReturn("Hello
World!");
        mockMvc.perform(get("/hello"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/plain"))
            .andExpect(content().string("Hello World!"));
```

```
        }
}
```

Testing with Running Server

- If a running server is needed, the usage of a random port is recommended
- Tests that need to make actual REST calls can use the **RestClient** and **AssertJ** for the result verification
- The server port can be injected using the `@LocalServerPort` annotation

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloTests {

    private RestClient restClient;

    public RestClient(@LocalServerPort int port) {
        restClient = RestClient.create("http://localhost:" + port);
    }

    @Test
    public void testHello() {
        ResponseEntity<String> response =
            restClient.get().uri("/hello").toEntity(String.class);
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);

assertThat(response.getHeaders().getContentType()).isEqualTo("text/pla
in");
```

```
        assertThat(response.getBody()).isEqualTo("Hello World!");
    }
}
```

Testing using REST Assured

- **REST Assured** is a library that simplifies the testing of REST based services and uses **Hamcrest** for result verification
- REST Assured uses the server `localhost` and port 8080 by default, but the port can easily be configured

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HelloTests {

    public HelloTests(@LocalServerPort int port) {
        RestAssured.port = port;
    }

    @Test
    public void testHello() {
        RestAssured.given().accept(ContentType.TEXT)
            .when().get("/hello")
            .then().statusCode(OK.value())
                   .assertThat().contentType("text/plain")
                   .body(equalTo("Hello World!"));
```

```
        }
}
```

Additional Features

Message Converters

- Spring MVC uses message converters to convert HTTP message bodies
- Objects can be automatically converted to and from JSON or XML (using the Jackson XML extension)
- Custom converters have to implement the **HttpMessageConverter** interface

```
public class StringMessageConverter<T> implements
HttpMessageConverter<T> {

    public List<MediaType> getSupportedMediaTypes() {
        return Collections.singletonList(MediaType.TEXT_PLAIN);
    }
    public boolean canWrite(Class<?> clazz, MediaType mediaType) {
        return true;
    }
    public void write(T object, MediaType contentType,
HttpOutputMessage message) {
        message.getBody().write(object.toString().getBytes());
    }
```

```
    ...
}
```

Adding Message Converters

- Message converters can be added by overriding the `configureMessageConverters` method of the `WebMvcConfigurer` interface

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureMessageConverters(
            List<HttpMessageConverter<?>> converters) {
        converters.add(new StringMessageConverter());
    }
}
```

Filters

- Filters encapsulate recurring tasks (e.g. logging, authentication) in reusable components
- Filters may examine, modify or exchange the request and response objects and even cancel the request processing
- Filters are part of the **Servlet specification** and have to implement the **Filter** interface
- Filters are invoked before the `DispatcherServlet`

```
@Component
@Order(1)
public class LoggingFilter extends HttpFilter {

    public void doFilter(HttpServletRequest request,
HttpServletResponse response,
                         FilterChain chain) throws IOException,
ServletException {
        // log request
        chain.doFilter(request, response);
        // log response
```

```
        }
}
```

Registering Filters

- Filters that are Spring beans, are automatically registered with the embedded container
- The `@Order` annotation can be used to define an order of multiple filters
- By default, filters are mapped to the URI `/*` and thus intercept all requests
- The filters can be configured more accurately using a `FilterRegistrationBean`

```
@Configuration
public class WebConfig {

    @Bean
    public FilterRegistrationBean<LoggingFilter> filterRegistration()
{
        FilterRegistrationBean<LoggingFilter> registration =
            new FilterRegistrationBean<>();
        registration.setFilter(new LoggingFilter());
        registration.addUrlPatterns("/api/*");
        return registration;
```

```
        }
}
```

Request Validation

- If the starter `spring-boot-starter-validation` is added to a Spring web application, **Bean Validation** of controller methods is enabled
- Validation of request bodies is activated by annotating the parameters with `@Valid` and their classes with appropriate validation constraints
- If the validation fails, a `MethodArgumentNotValidException` is thrown and an error response 400 (Bad Request) is returned

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @PostMapping
    public void registerCustomer(@RequestBody @Valid Customer
customer) {
        ...
    }
}
```

```
public class Customer {
    @NotEmpty private String name;
    @Email private String email;
    ...
}
```

Request Validation (cont.)

- Validation of path variables and request parameters is activated by annotating the controller with `@Validated` and the method parameters with appropriate validation constraints
- If the validation fails, a `ConstraintViolationException` is thrown and an error response 500 (Internal Server) is returned

```
@RestController
@RequestMapping("/customers")
@Validated
public class CustomerController {

    @GetMapping("/{id}/orders")
    public List<Order> findOrders(@PathVariable @Positive long id,
                                  @RequestParam @Min(1980) Integer
year) {
        ...
```

```
        }
}
```

Programmatic Validation

- There may be cases when programmatic validation of objects is required
- For this purpose, a validator can be injected and used to validate the objects

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @Autowired
    private Validator validator;

    @PostMapping
    public void registerCustomer(@RequestBody Customer customer) {
        Set<ConstraintViolation<Input>> violations =
validator.validate(customer);
        if (!violations.isEmpty()) {
            ProblemDetail detail = ...
            throw new ErrorResponseException(HttpStatus.BAD_REQUEST,
detail, null);
        }
        ...
```

```
        }
}
```

CORS Support

- **Cross-origin resource sharing** (CORS) is a W3C specification that determines which cross-domain requests are authorized
- Spring MVC supports CORS by annotating controllers with the `@CrossOrigin` annotation

```
@RestController
@RequestMapping("/customers")
@CrossOrigin(origins = "http://example.org", methods = GET)
public class CustomerController {
    ...
}
```

CORS Support (cont.)

- A global CORS configuration for all controllers can be defined by registering a `WebMvcConfigurer` bean

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
            public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOriginPatterns("http://example.org")
                .allowedMethods("GET");
    }
}
```

OpenAPI Documentation

- **OpenAPI** (formerly known as Swagger) is a specification for describing and visualizing RESTful web services
- **SpringDoc** is a library for automatic generation of OpenAPI documentation of Spring Boot projects
- SpringDoc examines an application at runtime to infer the API semantics based on Spring configurations, class structure and annotations
- In order to generate the OpenAPI documentation, the following dependency has to be added to the project

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.8.5</version>
</dependency>
```

OpenAPI Document Generation

- SpringDoc examines an application at runtime to infer the API semantics based on Spring configurations, class structure and annotations
- If the API includes bean validation annotations, SpringDoc generates schema documentation for the corresponding constraints
- **OpenAPI annotations** can be used to add additional descriptions

```
@RestController
@RequestMapping("/customers")
public class CustomerController {

    @Operation(summary = "Register a customer")
    @ApiResponse(responseCode = "201", description = "Customer
registered")
    @ApiResponse(responseCode = "409", description = "Customer already
exists")
    public void registerCustomer(@RequestBody Customer customer) {
        ...
```

```
        }
}
```

OpenAPI Configuration

- The generated OpenAPI specification is available at `/v3/api-docs` and the corresponding Swagger user interface at `/swagger-ui.html`
- Both locations can be redefined using the following application properties

```
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/api-docs-ui.html
```

Monitoring

Spring Boot Actuator

- Spring Boot includes a number of production-ready features that allow an application to be monitored and managed by using HTTP endpoints or JMX
- In addition, auditing, health, and metrics gathering can be applied to an application
- To enable the features, a dependency to the `spring-boot-starter-actuator` starter has to be added

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

Built-in Endpoints

ID	Description
`beans`	Displays a complete list of all the Spring beans in the application
`configprops`	Displays a list of all configuration properties
`env`	Exposes properties from Spring's configurable environment
`health`	Shows application health information
`httptrace`	Displays HTTP trace information
`info`	Displays arbitrary application info
`loggers`	Shows and modifies the configuration of loggers in the application
`metrics`	Shows metrics information for the current application
`mappings`	Displays a collated list of all request mapping paths

ID	Description
`shutdown`	Lets the application be gracefully shutdown.

Enabling and Exposing Endpoints

- By default, all endpoints except for `shutdown` are enabled
- To enable or disable an endpoint, the `management.endpoint.<id>.enabled` property can be used

```
management.endpoint.shutdown.enabled=true
```

- Since endpoints may contain sensitive information, only `health` and `info` are exposed as HTTP endpoints
- To define which endpoints are exposed, the `management.endpoints.web.exposure` property can be used

```
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans
```

- In addition, the HTTP endpoints are secured, if Spring security is present

Customizing the Endpoints

- By default, the management endpoints are exposed at the same port as the application
- A specific endpoint can be invoked by using its ID with the prefix `/actuator` as path, e.g. `/actuator/health`
- Port, prefix and path can be changed with corresponding application properties

```
management.server.port=8081
management.endpoints.web.base-path=/manage
management.endpoints.web.path-mapping.configprops=config
```