
Pengenalan MVC

Sejarah MVC

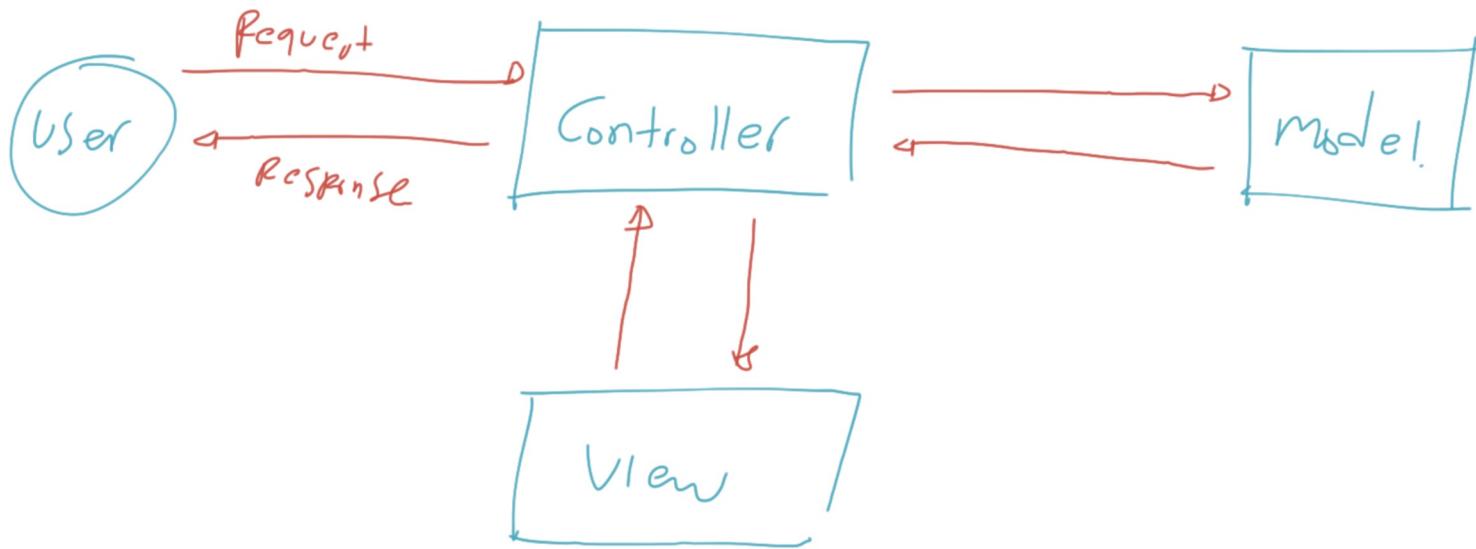
- MVC singkatan dari Model View Controller, yaitu salah satu software design pattern yang banyak digunakan ketika pengembangan aplikasi berbasis user interface
- MVC pertama kali dikenalkan oleh Trygve Reenskaug pada tahun 1970 ketika berkunjung ke Xerox Palo Alto Research
- Awalnya MVC banyak digunakan di aplikasi berbasis Desktop, namun sekarang MVC banyak diadopsi di Web
- Saat ini sendiri, design pattern MVC sudah banyak berkembang, ada hierarchical model-view-controller (HMVC), model-view-adapter (MVA), model-view-presenter (MVP), model-view-viewmodel (MVVM), dan lain-lain

Model View Controller

Seperti singkatannya, MVC dibagi menjadi tiga bagian :

- Model, merupakan bagian yang merepresentasikan data. Seperti yang kita ketahui, ada banyak sekali jenis data, seperti data request, data response, data table, dan lain-lain, sehingga kadang kita perlu memperkecil lagi scope dari Model itu sendiri ketika membuat aplikasi.
- View, merupakan bagian yang merepresentasikan tampilan, seperti halaman web, desktop, mobile, dan lain-lain.
- Controller, merupakan bagian yang mengurus alur kerja dari menerima input, memanipulasi data Model, sampai menampilkan View. Anggap saja Controller merupakan core logic dari aplikasi kita

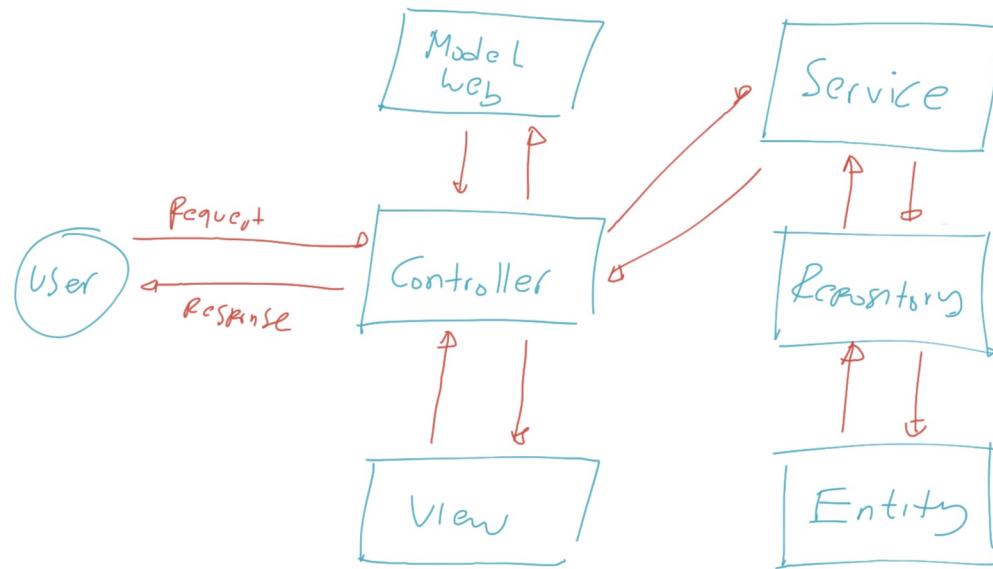
Diagram MVC



Pada Kenyataannya

- Walaupun sekilas MVC sangat sederhana, pada kenyataannya ketika kita membuat aplikasi yang kompleks, kita biasanya tidak lagi bisa memanfaatkan MVC
- Kadang kita butuh mengimplementasikan design pattern lain, seperti misal nya Service Pattern, Repository Pattern, dan lain-lain
- Oleh karena itu, jangan terlalu terpaku pada satu pattern, jika kita bisa mengkombinasikan beberapa pattern agar kode aplikasi kita lebih rapi dan baik, maka disarankan untuk melakukan kombinasi

Diagram Contoh Aplikasi



Pengenalan Spring Web MVC

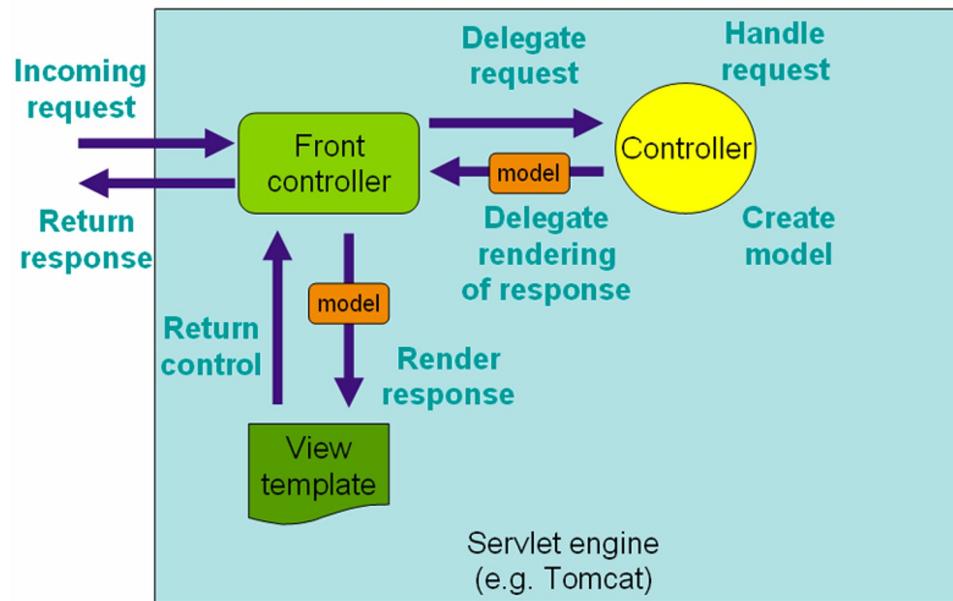
Pengenalan Spring Web MVC

- Spring Web MVC (Model View Controller) adalah sebuah fitur di Spring untuk mempermudah membuat web menggunakan Java Servlet
- Pada kelas Java Servlet kita sudah tahu bagaimana sulitnya membuat web di Java Servlet karena semua harus dibuat secara manual
- Spring membuat fitur Web MVC yang bisa digunakan untuk mempermudah semua proses pembuatan Web

Dispatcher Servlet

- Semua logic Spring Web MVC, diatur oleh sebuah servlet bernama DispatcherServlet
- Servlet ini adalah gerbang utama masuknya request di Spring Web MVC
- Dari DispatcherServlet, nanti akan diteruskan ke Controller yang sesuai dengan URL yang diakses
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html>

Cara Kerja Spring Web MVC



Membuat Project

Membuat Project

- <https://start.spring.io/>
- Tambah Dependency :
 - Spring Web MVC
 - Mustache
 - Lombok
 - Validation
 - Configuration Properties

Controller

Controller

- Untuk membuat Controller di Spring, kita bisa menggunakan annotation Controller
- Di annotation Controller sendiri, sebenarnya terdapat annotation Component, hal ini membuat class yang kita tambahkan annotation Controller, akan secara otomatis teregistrasi sebagai Bean
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Controller.html>



Kode : Controller

```
© HelloController.java ×  
S 1 package programmerzamannow.webmvc.controller;  
2  
3 import org.springframework.stereotype.Controller;  
4  
5 @Controller  
6 Q public class HelloController {  
7 }  
8 |
```

Request Mapping

Request Mapping

- Saat kita belajar menggunakan Servlet, untuk membuat Routing pada Servlet kita menggunakan annotation WebServlet
- Di Spring WebMVC, untuk menambahkan Routing, kita bisa menggunakan annotation RequestMapping pada method yang ingin kita jadikan sebagai Controller Handler nya
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html>



Kode : Request Mapping

```
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import java.io.IOException;

@Controller
public class HelloController {

    @RequestMapping(path = "/hello")
    public void helloWorld(HttpServletResponse response) throws IOException {
        response.getWriter().println("Hello World");
    }
}
```

Menjalankan Web

Menjalankan Web

- Spring Boot secara default menambahkan Apache Tomcat sebagai Embedded Web Server
- Hal ini menjadikan kita tidak perlu lagi untuk membuat aplikasi Spring Boot dalam bentuk War, dan tidak perlu melakukan deployment secara manual ke Apache Tomcat
- Secara default, Spring Boot menggunakan port 8080 untuk menjalankan Apache Tomcat nya
- Jika kita ingin mengubah port nya, kita bisa gunakan properties
- server.port=NOMOR
- Pada application.properties

Tampilan Web Browser



Servlet Request dan Response

Servlet Request dan Response

- Saat kita membuat Controller Handler dengan RequestMapping
- Kita bisa menambahkan parameter HttpServletRequest atau HttpServletResponse jika memang butuh object tersebut
- Tidak ada aturan posisi parameter, karena Spring WebMVC bisa mendeteksi secara otomatis tipe dan posisi parameter nya

Kode : Servlet Request dan Response

```
@Controller
public class HelloController {

    @RequestMapping(path = "/hello")
    public void helloWorld(HttpServletRequest request, HttpServletResponse response) throws IOException {
        String name = request.getParameter("name");
        if (Objects.isNull(name)) {
            name = "Guest";
        }
        response.getWriter().println("Hello " + name);
    }
}
```

Mock MVC

MockMVC

- Saat kita membuat Web menggunakan Spring WebMVC, Spring telah menyediakan fitur bernama MockMVC
- Fitur ini digunakan untuk mempermudah kita melakukan unit test
- Dengan menggunakan MockMVC, kita bisa mengetes semua Controller yang kita buat, tanpa harus menjalankan aplikasi Web nya, dan tidak perlu melakukan pengetesan secara manual menggunakan Browser atau HTTP Client
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html>

Static Imports

Ketika menggunakan MockMVC, kita butuh beberapa static utility method dari class-class berikut

- `MockMvcBuilders.*`
- `MockMvcRequestBuilders.*`
- `MockMvcResultMatchers.*`
- `MockMvcResultHandlers.*`

Kode : Membuat Mock MVC

```
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
5 import org.springframework.boot.test.context.SpringBootTest;
6 import org.springframework.test.web.servlet.MockMvc;
7
8 import static org.springframework.test.web.servlet.MockMvcBuilder.*;
9 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
10 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
11 import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;
12
13 no usages
14
15 @SpringBootTest
16 @AutoConfigureMockMvc
17 public class HelloControllerTest {
18
19     @Autowired
20     private MockMvc mockMvc;
```



Kode : Unit Test Hello Route

```
@Test
void helloGuest() throws Exception {
    mockMvc.perform(
        get("/hello")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello Guest"))
    );
}

@Test
void helloName() throws Exception {
    mockMvc.perform(
        get("/hello").queryParam("name", "Eko")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello Eko"))
    );
}
```

Integration Test

Integration Test

- Saat kita menggunakan MockMVC, Spring tidak akan menjalankan aplikasi web kita
- Spring hanya menyediakan mock request da mock response
- Test yang mensimulasikan saat aplikasi berjalan adalah menggunakan mode Integration Test
- Integration Test artinya adalah menjalankan aplikasi web secara lengkap, bersama dengan web server nya (Apache Tomcat)
- Secara otomatis kita bisa menjalankan aplikasi web ketika test berjalan, dan menghentikannya ketika test selesai



Kode : Menjalankan Integration Test

```
no usages
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerIntegrationTest {

    @Autowired
    private TestRestTemplate restTemplate;

}
```

Test Rest Template

- Berbeda ketika kita menggunakan MockMVC, saat menggunakan mode Integration Test, karena tidak menggunakan mock lagi, maka untuk mengetest aplikasi, kita harus benar-benar mengirim request ke aplikasi web
- Spring memiliki HTTP Client bernama RestTemplate, yang akan kita bahas di materi khusus
- Dan spesial untuk integration test, kita bisa menggunakan object TestRestTemplate



Random Port

- Secara default, saat menjalankan Integration Test, Spring akan menjalankan aplikasi sesuai dengan port di properties server.port
- Namun kadang-kadang, portnya bentrok dengan port lain, oleh karena itu direkomendasikan menggunakan random port
- Random port artinya Spring akan mencoba mendeteksi port yang belum digunakan, nanti secara otomatis akan menggunakan port tersebut
- Untuk mendapatkan nilai port nya, kita bisa menggunakan inject @Value("\${local.server.port}") atau lebih mudah menggunakan @LocalServerPort



Kode : Random Port

```
no usages
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerIntegrationTest {

    @LocalServerPort
    private Integer port;

    @Autowired
    private TestRestTemplate restTemplate;

}
```



Kode : Integration Test

```
@Test
void helloGuest() {
    String response = restTemplate.getForEntity("http://localhost:" + port + "/hello", String.class)
        .getBody();
    Assertions.assertNotNull(response);
    Assertions.assertEquals("Hello Guest", response.trim());
}

@Test
void helloEko() {
    String response = restTemplate.getForEntity("http://localhost:" + port + "/hello?name=Eko", String.class)
        .getBody();
    Assertions.assertNotNull(response);
    Assertions.assertEquals("Hello Eko", response.trim());
}
```

Service

Service Layer

- Di awal kita belajar tentang MVC (Model View Controller)
- Di bahasa pemrograman atau framework lain, biasanya orang menambahkan kode yang berhubungan dengan bisnis logic di Controller Layer, namun berbeda dengan programmer Java
- Untuk programmer Java, sebenarnya kebiasaan atau best practice nya akan membuat layer khusus untuk kode bisnis logic, bernama Service Layer
- Service Layer di Spring memiliki annotation khusus, yaitu `@Service`
- Saat kita menambahkan `@Service`, secara otomatis juga class tersebut akan di registrasikan sebagai bean
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html>



Interface

- Salah satu best practice di Spring adalah, saat kita membuat Service Layer, kita akan buat dalam bentuk Interface
- Lalu kita akan buat class implementasi yang diregistrasikan sebagai Spring Bean
- Sedangkan class yang membutuhkan Service Layer tersebut, akan menggunakan Interface nya, bukan class implementasinya
- Salah satu keuntungan mengekspos Interface dibanding Class adalah, kita bisa mengubah atau mengganti isi dari class implementasi, tanpa berdampak pada class lain yang menggunakan interface nya

Kode : Hello Service

```
no usages
public interface HelloService {
    no usages
    String hello(String name);
}
```

```
@Service
public class HelloServiceImpl implements HelloService {
    no usages
    @Override
    public String hello(String name) {
        if (name == null) {
            return "Hello Guest";
        } else {
            return "Hello " + name;
        }
    }
}
```



Kode : Hello Service Test

```
@SpringBootTest
class HelloServiceImplTest {

    @Autowired
    private HelloService helloService;

    @Test
    void hello() {
        Assertions.assertEquals("Hello Guest", helloService.hello(null));
        Assertions.assertEquals("Hello Eko", helloService.hello("Eko"));
    }
}
```

Mock Bean



Mock Bean

- Saat kita belajar di kelas Java Unit Test, kita sudah belajar tentang melakukan mock menggunakan Mockito
- Saat kita menggunakan Spring, kita juga melakukan hal tersebut
- Selain itu, Spring juga bisa secara otomatis meregistrasikan Mock object tersebut sebagai bean, sehingga class yang membutuhkan bean tersebut, secara otomatis bisa mendapatkan Mock object yang kita buat
- Untuk membuat Mock Bean, kita cukup gunakan annotation @MockBean
- <https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html>



Kode : Hello Controller

```
@Controller
public class HelloController {

    @Autowired
    private HelloService helloService;

    @RequestMapping(path = @v"/hello")
    public void helloWorld(HttpServletRequest request, HttpServletResponse response) throws IOException {
        String name = request.getParameter("name");
        String responseBody = helloService.hello(name);
        response.getWriter().println(responseBody);
    }
}
```



Kode : Mock Bean

```
@MockBean
private HelloService helloService;

@BeforeEach
void setUp() {
    Mockito.when(helloService.hello(Mockito.anyString()))
        .thenReturn("Hello Guys");
}

@Test
void helloGuest() throws Exception {
    mockMvc.perform(
        get("/hello")
            .queryParam("name", "Budi")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello Guys"))
    );
}
```

Request Method

Request Method

- Saat kita menggunakan RequestMapping, terdapat attribute method yang bisa kita gunakan untuk menentukan jenis HTTP Method yang diperbolehkan
- Secara default, jika kita tidak memilihnya, maka Controller Method tersebut bisa diakses oleh seluruh jenis HTTP Method
- Jika kita mengirim method yang tidak diperbolehkan, maka Spring akan menolak dengan response 405 Method Not Allowed



Kode : Request Method GET

```
@Controller
public class HelloController {

    @Autowired
    private HelloService helloService;

    @RequestMapping(path = @v"/hello", method = RequestMethod.GET)
    public void helloWorld(HttpServletRequest request, HttpServletResponse response) throws IOException {
        String name = request.getParameter("name");
        String responseBody = helloService.hello(name);
        response.getWriter().println(responseBody);
    }
}
```



Kode : Unit Test Post

```
@Test
void helloPost() throws Exception {
    mockMvc.perform(
        post("/hello")
            .queryParam("name", "Eko")
    ).andExpectAll(
        status().isMethodNotAllowed()
    );
}
```



Shortcut Annotation

RequestMapping Method	Shortcut Annotation
GET	@GetMapping
POST	@PostMapping
PUT	@PutMapping
PATCH	@PatchMapping
DELETE	@DeleteMapping



Kode : Shortcut Mapping

```
@Controller
public class HelloController {

    @Autowired
    private HelloService helloService;

    @GetMapping(path = "/hello")
    public void helloWorld(HttpServletRequest request, HttpServletResponse response) throws IOException {
        String name = request.getParameter("name");
        String responseBody = helloService.hello(name);
        response.getWriter().println(responseBody);
    }
}
```

Request Param

Request Param

- Saat kita belajar Java Servlet, kita sudah tahu untuk mendapatkan Query/Request Parameter dari ServletRequest
- Namun di Spring, kita bisa menggunakan Annotation @RequestParam untuk memberitahu bahwa kita membutuhkan request parameter
- Selain itu, kita bisa menambahkan apakah query parameter itu wajib atau tidak, dan juga bisa menambahkan default value nya jika tidak dikirim oleh user
- Secara otomatis data request parameter akan dikirim datanya ke parameter yang kita tentukan
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam.html>



Kode : Request Param

```
@Controller
public class HelloController {

    @Autowired
    private HelloService helloService;

    @GetMapping(path = @"/hello")
    public void helloWorld(@RequestParam(name = "name", required = false) String name,
                          HttpServletResponse response) throws IOException {
        String responseBody = helloService.hello(name);
        response.getWriter().println(responseBody);
    }
}
```

Konversi Tipe Data

- Kita tahu bahwa query parameter itu datanya adalah String
- Namun jika kita membutuhkan datanya dalam bentuk tipe data lain, Spring bisa secara otomatis melakukan konversi tipe datanya menggunakan fitur Converter yang pernah kita bahas di materi Spring Config Properties



Kode : Membuat Date Converter

```
✓ @Component
  @Slf4j
public class StringToDateConverter implements Converter<String, Date> {

    1 usage
    private SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

    @Override
    public Date convert(String source) {
        try {
            return dateFormat.parse(source);
        } catch (ParseException e) {
            log.warn("Error convert date from string {}", source, e);
            return null;
        }
    }
}
```



Kode : Date Controller

```
@Controller
public class DateController {

    1 usage

    private final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

    @GetMapping(path = "/date")
    public void getDate(@RequestParam(name = "date") Date date,
                        HttpServletResponse response) throws IOException {
        response.getWriter().println("Date : " + dateFormat.format(date));
    }
}
```



Kode : Date Controller Test

```
@Test
void date() throws Exception {
    mockMvc.perform(
        get("/date")
            .queryParam("date", "2020-10-10")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Date : 20201010"))
    );
}
```

Response Body

Response Body

- Secara default, kita harus menuliskan response dari Controller Method ke HttpServletResponse
- Namun hal ini kadang menyulitkan jika misal kita hanya ingin mengembalikan data berupa String
- Spring memiliki annotation @ResponseBody, yang bisa secara otomatis menjadikan data yang dikembalikan dari Controller Method menjadi data yang ditulis ke HttpServletResponse
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseBody.html>



Kode : Date Controller

```
@Controller
public class DateController {

    1 usage

    private final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyyMMdd");

    @GetMapping(path = @v"/date")
    @ResponseBody
    public String getDate(@RequestParam(name = "date") Date date) {
        return "Date : " + dateFormat.format(date);
    }
}
```

Request Content Type

Request Content Type

- Saat kita membuat Controller Method, kita juga bisa membatasi jenis Content-Type yang dikirim oleh user
- Contoh pada kasus melakukan submit data form, kita biasanya meminta Content-Type yang dikirim oleh user adalah application/x-www-form-urlencoded
- Untuk membatasi tipe Content-Type, kita bisa tambahkan di @RequestMapping pada attribute consume

Kode : Form Controller

```
@Controller
public class FormController {

    @PostMapping(path = @"/form/hello", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
    @ResponseBody
    public String hello(@RequestParam(name = "name") String name) {
        return "Hello " + name;
    }

}
```



Kode : Form Controller Unit Test

```
@Test
void formHello() throws Exception {
    mockMvc.perform(
        post("/form/hello")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("name", "Eko")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello Eko"))
    );
}
```

Response Content Type

Response Content Type

- Di @RequestMapping, selain consume, terdapat juga attribute produce, yang bisa kita gunakan untuk memberi tahu di HTTP Response, Content-Type dari response body yang dikembalikan

Kode : Response Content Type

```
@PostMapping(  
    path = @"/form/hello",  
    consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE,  
    produces = MediaType.TEXT_HTML_VALUE  
)  
  
@ResponseBody  
public String hello(@RequestParam(name = "name") String name) {  
    return """  
        <html>  
        <body>  
        <h1>Hello ${name}</h1>  
        </body>  
        </html>  
    """ .replace("${name}", name);  
}
```



Kode : Form Controller Unit Test

```
@Test
void formHello() throws Exception {
    mockMvc.perform(
        post("/form/hello")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("name", "Eko")
    ).andExpectAll(
        status().isOk(),
        header().string(HttpHeaders.CONTENT_TYPE, Matchers.containsString(MediaType.TEXT_HTML_VALUE)),
        content().string(Matchers.containsString("Hello Eko"))
    );
}
```

Request Header

Request Header

- Untuk mendapatkan Http Request Header, seperti yang sudah kita pelajari di materi Java Servlet, kita bisa mendapatkannya melalui HttpServletRequest
- Namun Spring WebMVC memiliki cara lebih mudah dengan menggunakan annotation @RequestHeader
- Caranya kita bisa tambahkan di parameter di Controller Method
- Kita juga bisa menentukan apakah wajib atau tidak, dan juga default value nya
- Selain itu, fitur Converter juga bisa digunakan untuk Request Header
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestHeader.html>



Kode : Header Controller

```
@Controller
public class HeaderController {

    @GetMapping(path = "/header/token")
    @ResponseBody
    public String header(@RequestHeader(name = "X-TOKEN") String token) {
        if (token.equals("OK")) {
            return "OK";
        } else {
            return "KO";
        }
    }
}
```



Kode : Header Unit Test

```
@Test
void headerOk() throws Exception {
    mockMvc.perform(
        get("/header/token")
            .header("X-TOKEN", "EKO")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("OK"))
    );
}
```

Path Variable

Path Variable

- Salah satu fitur Spring WebMVC yang berbeda dari Java Servlet adalah Path Variable
- Path Variable adalah fitur dimana kita bisa membuat patterns pada URL Path, dan mengambil nilai yang terdapat di URL Path nya
- Dengan fitur ini, kita bisa membuat URL Path yang dinamis, dan bisa mendapatkan nilai dinamis di URL Path nya secara otomatis
- Untuk menggunakan fitur ini, kita perlu tambahkan variable path nya di URL Path nya, dan juga menambahkan parameter dengan annotation @PathVariable
- Path Variable juga memiliki kemampuan otomatis konversi tipe data dengan Converter
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVariable.html>



Kode : Order Controller

```
@Controller
public class OrderController {

    @GetMapping(path = "/orders/{orderId}/products/{productId}")
    @ResponseBody
    public String order(
        @PathVariable("orderId") String orderId,
        @PathVariable("productId") String productId
    ) {
        return "Order : " + orderId + ", Product : " + productId;
    }
}
```



Kode : Unit Test Order Controller

```
@Test
void orderProduct() throws Exception {
    mockMvc.perform(
        get("/orders/1/products/2")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Order : 1, Product : 2"))
    );
}
```

Form Request

Form Request

- Seperti pernah kita bahas di materi Java Servlet, untuk mendapatkan data Form Request, kita bisa menggunakan cara yang sama dengan mendapatkan data di Query Parameter
- Begitu pula di Spring Web MVC
- Untuk mendapatkan data di Form Request, kita bisa menggunakan annotation @RequestParam
- Secara otomatis Spring Web MVC juga akan mengambil data dari Form Request atau Query Parameter
- Seperti yang pernah kita praktekan di materi Request Content Type



Kode : Form Controller

```
1 usage
private final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

@PostMapping(path = "/form/person", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
@ResponseBody
public String createPerson(
    @RequestParam(name = "name") String name,
    @RequestParam(name = "birthDate") Date birthDate,
    @RequestParam(name = "address") String address
) {
    return "Success create Person with name : " + name +
        ", birthDate : " + dateFormat.format(birthDate) +
        ", address : " + address;
}
```



Kode : Form Controller Test

```
@Test
void createPerson() throws Exception {
    mockMvc.perform(
        post("/form/person")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("name", "Eko")
            .param("birthDate", "1990-10-10")
            .param("address", "Indonesia")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Success create Person with name : Eko, " +
            "birthDate : 1990-10-10, " +
            "address : Indonesia"))
    );
}
```

Upload File

Upload File

- Upload File di Spring Web MVC bisa menggunakan cara seperti di Java Web Servlet, atau bisa menggunakan fitur di Spring Web MVC yang lebih mudah menggunakan annotation @RequestPart
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestPart.html>
- Untuk tipe data pada parameter nya, kita bisa gunakan MultipartFile
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html>

Upload File Properties

- Spring memiliki pengaturan yang bisa kita atur untuk Upload file, misal kita ingin membatasi jumlah ukuran file misal nya, dan lain-lain
- Semua pengaturan untuk upload file bisa kita tambahkan di application properties, dengan prefix `spring.servlet.multipart`
- Kita bisa lihat daftarnya disini :
- <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.web>



Kode : Upload Controller

```
@Controller
public class UploadController {

    @PostMapping(path = @v"/upload/profile", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    @ResponseBody
    public String upload(@RequestParam(name = "name") String name,
                         @RequestPart(name = "profile") MultipartFile profile) throws IOException {
        Path path = Path.of("upload/" + profile.getOriginalFilename());
        Files.write(path, profile.getBytes());
        return "Success save profile " + name + " to " + path;
    }
}
```

Kode : Test Upload Controller

```
@Test
void uploadFile() throws Exception {
    mockMvc.perform(
        multipart("/upload/profile") MockMultipartHttpServletRequest...
            .file(new MockMultipartFile("profile", "profile.png", "image/png",
                getClass().getResourceAsStream("/images/profile.png")))
            .param("name", "Eko") MockHttpRequestBuilder
            .contentType(MediaType.MULTIPART_FORM_DATA)
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Success save profile Eko to upload/profile.png"))
    );
}
```

Request Body

Request Body

- Saat kita membuat aplikasi web berupa RESTful API, kadang kita ingin mengirim data lewat Request Body dalam bentuk format data seperti JSON, XML, dan sejenisnya
- Spring bisa digunakan untuk membaca data Request Body secara mudah, cukup menggunakan annotation `@RequestBody`
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestBody.html>

Kode : Hello Request & Response

```
no usages
public class HelloRequest {

    2 usages
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
no usages
public class HelloResponse {

    2 usages
    private String hello;

    public String getHello() {
        return hello;
    }

    public void setHello(String hello) {
        this.hello = hello;
    }
}
```



Kode : Body Controller

```
@PostMapping(  
    path = @v"/body/hello",  
    consumes = MediaType.APPLICATION_JSON_VALUE,  
    produces = MediaType.APPLICATION_JSON_VALUE  
)  
  
@ResponseBody  
public String body(@RequestBody String requestBody) throws JsonProcessingException {  
    HelloRequest request = objectMapper.readValue(requestBody, HelloRequest.class);  
  
    HelloResponse response = new HelloResponse();  
    response.setHello("Hello " + request.getName());  
  
    return objectMapper.writeValueAsString(response);  
}
```



Kode : Unit Test Body Controller

```
void bodyHello() throws Exception {
    HelloRequest request = new HelloRequest();
    request.setName("Eko");

    mockMvc.perform(
        post("/body/hello")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(request)))
        .andExpectAll(
            status().isOk())
        .andExpect(result -> {
            String responseBody = result.getResponse().getContentAsString();
            HelloResponse helloResponse = objectMapper.readValue(responseBody, HelloResponse.class);
            Assertions.assertEquals("Hello Eko", helloResponse.getHello());
        });
}
```

Response Status

Response Status

- Saat kita membuat HTTP Response, kadang kita ingin mengubah Response Status Code
- Secara default, response sukses adalah 200, kadang mungkin kita ingin ubah secara manual
- Jika kita ingin ubah secara dinamis, kita bisa gunakan HttpServletResponse
- Atau jika kita ingin hardcode response status nya, kita bisa gunakan annotation @ResponseStatus di Controller Method nya
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ResponseStatus.html>



Kode : Code Controller

```
@Controller
public class CodeController {

    @DeleteMapping(path = @"/products/{id}")
    @ResponseBody
    @ResponseStatus(HttpStatus.ACCEPTED)
    public void delete(@PathVariable("id") Integer id) {
        // delete to database
    }

}
```



Kode : Test Code Controller

```
@Test
void deleteProduct() throws Exception {
    mockMvc.perform(
        delete("/products/12345")
    ).andExpectAll(
        status().isAccepted()
    );
}
```

Response Entity

Response Entity

- Sekarang kita sudah tahu beberapa cara membuat HTTP Response, dari menggunakan HttpServletResponse dan @ResponseBody
- Spring menyediakan cara yang sangat flexible untuk membuat HTTP Response menggunakan object ResponseEntity
- Kita bisa return di Controller Method dengan object ResponseEntity
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>



Kode : Auth Controller

```
@Controller
public class AuthController {

    @PostMapping(path = @v"/auth/login", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
    public ResponseEntity<String> login(@RequestParam(name = "username") String username,
                                         @RequestParam(name = "password") String password) {

        if ("eko".equals(username) && "rahasia".equals(password)) {
            return new ResponseEntity<>("OK", HttpStatus.OK);
        } else {
            return new ResponseEntity<>("KO", HttpStatus.UNAUTHORIZED);
        }
    }
}
```



Kode : Unit Test Sukses

```
@Test
void loginSuccess() throws Exception {
    mockMvc.perform(
        post("/auth/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("username", "eko")
            .param("password", "rahasia")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("OK"))
    );
}
```



Kode : Unit Test Gagal

```
@Test
void loginFailed() throws Exception {
    mockMvc.perform(
        post("/auth/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("username", "eko")
            .param("password", "salah")
    ).andExpectAll(
        status().isUnauthorized(),
        content().string(Matchers.containsString("KO"))
    );
}
```

Cookie



Cookie

- Cara membuat Cookie di Spring Web MVC bisa dilakukan dengan menggunakan HttpServletResponse seperti pada Java Servlet
- Namun untuk membaca Cookie yang dikirim oleh Web Browser, kita bisa otomatis menggunakan annotation @CookieValue
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/CookieValue.html>



Kode : Auth Controller Login

```
@PostMapping(path = @"/auth/login", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public ResponseEntity<String> login(@RequestParam(name = "username") String username,
                                    @RequestParam(name = "password") String password,
                                    HttpServletResponse servletResponse) {

    if ("eko".equals(username) && "rahasia".equals(password)) {
        Cookie cookie = new Cookie("username", username);
        cookie.setPath("/");
        servletResponse.addCookie(cookie);
        return new ResponseEntity<>("OK", HttpStatus.OK);
    } else {
        return new ResponseEntity<>("KO", HttpStatus.UNAUTHORIZED);
    }
}
```



Kode : Test Auth Controller Login

```
@Test
void loginSuccess() throws Exception {
    mockMvc.perform(
        post("/auth/login")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("username", "eko")
            .param("password", "rahasia")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("OK")),
        cookie().value("username", Matchers.is("eko"))
    );
}
```



Kode : Auth Controller Get User

```
@GetMapping(path = @"/auth/user")
public ResponseEntity<String> getUser(@CookieValue("username") String username) {
    return new ResponseEntity<>("Hello " + username, HttpStatus.OK);
}
```



Kode : Test Auth Controller Get User

```
@Test
void getUser() throws Exception {
    mockMvc.perform(
        get("/auth/user")
            .cookie(new Cookie("username", "eko"))
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello eko"))
    );
}
```

Model Attribute

Model Attribute

- Saat kita mengirim request berupa form dengan input yang banyak, kadang menyulitkan kita jika kita harus membuat semua parameter input dengan @RequestParam
- Bayangkan jika ada 10 input, maka kita harus membuat 10 parameter @RequestParam
- Spring memiliki fitur dimana kita bisa melakukan binding attribute yang dikirim dengan class Java Bean yang kita buat menggunakan annotation @ModelAttribute
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html>



Kode : Class Create Person Request

```
no usages
3 public class CreatePersonRequest {
4
5     2 usages
6     private String firstName;
7
8     2 usages
9     private String middleName;
0
1     2 usages
2     private String lastName;
3
4     2 usages
5     private String email;
6
7     2 usages
8     private String phone;
```



Kode : Person Controller

```
@PostMapping(path = "/person", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public String createPerson(@ModelAttribute CreatePersonRequest request) {
    return new StringBuilder().append("Success create person ")
        .append(request.getFirstName()).append(" ")
        .append(request.getMiddleName()).append(" ")
        .append(request.getLastName())
        .append(" with email ").append(request.getEmail())
        .append(" and phone ").append(request.getPhone())
        .toString();
}
```

Kode : Test Person Controller

```
@Test
void createPerson() throws Exception {
    mockMvc.perform(
        post("/person")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("firstName", "Eko")
            .param("middleName", "Kurniawan")
            .param("lastName", "Khannedy")
            .param("email", "eko@example.com")
            .param("phone", "080989999")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Success create person Eko Kurniawan Khannedy " +
            "with email eko@example.com and phone 080989999"))
    );
}
```

Nested Model

- Salah satu yang powerfull di Model Attribute adalah, kita bisa otomatis juga membuat object dari nested attribute di Model
- Misal pada kasus Person sebelumnya, misal saja terdapat sebuah attribute Address yang merupakan Java Bean lainnya
- Untuk mengisi data Address, kita bisa gunakan . (titik), misal address.street, address.city, dan seterusnya

Kode : Class Create Address Request

```
no usages  
3   public class CreateAddressRequest {  
4  
5       2 usages  
6           private String street;  
7  
8       2 usages  
9           private String city;  
0  
1       2 usages  
2           private String country;  
3  
4       2 usages  
5           private String postalCode;
```

```
2 usages  
3   public class CreatePersonRequest {  
4  
5       no usages  
6           private CreateAddressRequest address;  
7  
8               2 usages  
9                   private String firstName;  
0  
1  
2               2 usages  
3                   private String middleName;  
4  
5               2 usages  
6                   private String lastName;  
7  
8               2 usages  
9                   private String email;  
0  
1  
2               2 usages  
3                   private String phone;
```



Kode : Person Controller

```
@PostMapping(path = @"/person", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public String createPerson(@ModelAttribute CreatePersonRequest request) {
    return new StringBuilder().append("Success create person ")
        .append(request.getFirstName()).append(" ")
        .append(request.getMiddleName()).append(" ")
        .append(request.getLastName())
        .append(" with email ").append(request.getEmail())
        .append(" and phone ").append(request.getPhone())
        .append(" with address ")
        .append(request.getAddress().getStreet()).append(", ")
        .append(request.getAddress().getCity()).append(", ")
        .append(request.getAddress().getCountry()).append(", ")
        .append(request.getAddress().getPostalCode())
        .toString();
```



Kode : Test Person Controller

```
1 post("/person")
2     .contentType(MediaType.APPLICATION_FORM_URLENCODED)
3     .param("firstName", "Eko")
4     .param("middleName", "Kurniawan")
5     .param("lastName", "Khannedy")
6     .param("email", "eko@example.com")
7     .param("phone", "080989999")
8     .param("address.street", "Jalan Belum Jadi")
9     .param("address.city", "Jakarta")
10    .param("address.country", "Indonesia")
11    .param("address.postalCode", "11111")
12
13    .andExpectAll(
14        status().isOk(),
15        content().string(Matchers.containsString("Success create person Eko Kurniawan Khannedy " +
16            "with email eko@example.com and phone 080989999 with address " +
17            "Jalan Belum Jadi, Jakarta, Indonesia, 11111"))
18    );
19
```

List

- Selain nested attribute, kita juga bisa menggunakan List sebagai model attribute, cara mengirim parameter nya cukup mudah
- Untuk list dengan tipe data primitive seperti String, Integer, dan sejenisnya, kita bisa gunakan parameter :
 - namaParam[0]=data1
 - namaParam[1]=data2
 - Dan seterusnya
- Untuk list dengan tipe data object lagi, kita bisa gunakan parameter :
 - namaParam[0].field1=data1
 - namaParam[0].field2=data2
 - namaParam[1].field1=data1
 - namaParam[1].field2=data2

Kode : Create Social Media Request

```
public class CreateSocialMediaRequest {  
  
    2 usages  
    private String name;  
  
    2 usages  
    private String location;
```

```
2 usages  
public class CreatePersonRequest {  
  
    no usages  
    private List<String> hobbies;  
  
    no usages  
    private List<CreateSocialMediaRequest> socialMedias;  
  
    2 usages  
    private CreateAddressRequest address;  
  
    2 usages
```



Kode : Test Person Controller

```
.param("email", "eko@example.com")
.param("phone", "080989999")
.param("address.street", "Jalan Belum Jadi")
.param("address.city", "Jakarta")
.param("address.country", "Indonesia")
.param("address.postalCode", "11111")
.param("hobbies[0]", "Coding")
.param("hobbies[1]", "Reading")
.param("socialMedias[0].name", "Facebook")
.param("socialMedias[0].location", "facebook.com/ProgrammerZamanNow")
).andExpectAll()
```

Json

Json

- Spring Web MVC terintegrasi dengan baik dengan library Jackson untuk menangani tipe data JSON, baik itu untuk consume dari Request Body atau produce ke Response Body
- Saat kita menggunakan consume dengan tipe data JSON atau produce dengan tipe data JSON, kita tidak perlu secara manual melakukan konversi dari object ke JSON String, hal itu sudah otomatis di handle oleh Jackson

Konfigurasi Jackson

- Kita tidak perlu membuat Bean Jackson secara manual lagi, karena itu sudah di handle oleh Spring Boot
- Jika kita butuh melakukan konfigurasi untuk Jackson, kita bisa menggunakan application properties
- Semua daftar konfigurasinya bisa kita gunakan dengan prefix `spring.jackson.`
- <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.json>



Kode : Person API Controller

```
@PostMapping(  
    path = @"/api/person",  
    consumes = MediaType.APPLICATION_JSON_VALUE,  
    produces = MediaType.APPLICATION_JSON_VALUE  
)  
@ResponseBody  
public CreatePersonRequest createPerson(@RequestBody CreatePersonRequest request) {  
    return request;  
}  
}
```



Kode : Test Person API Controller

```
request.setHobbies(List.of("Coding", "Reading"));
request.setSocialMedias(new ArrayList<>());
request.getSocialMedias().add(new CreateSocialMediaRequest("Facebook", "facebook.com/ProgrammerZamanNow"));

mockMvc.perform(
    post("/api/person")
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(request)))
.andExpectAll(
    status().isOk(),
    content().json(objectMapper.writeValueAsString(request))
);
```

Validation

Validation

- Spring WebMVC terintegrasi dengan baik dengan Bean Validation seperti yang sudah kita bahas di materi Spring Validation
- Saat kita membuat parameter @ModelAttribute atau @RequestBody, jika object tersebut ingin di validasi secara otomatis menggunakan Bean Validation, kita bisa tambahkan annotation @Valid
- Jika data tidak valid, secara otomatis Spring akan mengembalikan response 400 Bad Request
- Khusus validasi di Controller, exception yang akan dibuat adalah
MethodArgumentNotValidException bukan ConstraintViolationException nya Bean Validation



Kode : Create Person Validation

```
13 package com.example.demo;
14
15 public class CreatePersonRequest {
16
17     private List<String> hobbies;
18
19     private List<CreateSocialMediaRequest> socialMedias;
20
21     private CreateAddressRequest address;
22
23     @NotBlank
24     private String firstName;
25
26     private String middleName;
27
28     private String lastName;
29
30     @NotBlank
31     private String email;
32
33     @NotBlank
34     private String phone;
35 }
```



Kode : Request Body Valid

```
@PostMapping(  
    path = @"/api/person",  
    consumes = MediaType.APPLICATION_JSON_VALUE,  
    produces = MediaType.APPLICATION_JSON_VALUE  
)  
@ResponseBody  
public CreatePersonRequest createPerson(@RequestBody @Valid CreatePersonRequest request) {  
    return request;  
}
```

Kode : Test Request Body Valid

```
void createPersonValidationError() throws Exception {
    CreatePersonRequest request = new CreatePersonRequest();
    request.setMiddleName("Kurniawan");
    request.setHobbies(List.of("Coding", "Reading"));
    request.setSocialMedias(new ArrayList<>());
    request.getSocialMedias().add(new CreateSocialMediaRequest("Facebook", "facebook.com/ProgrammerZamanNow"));

    mockMvc.perform(
        post("/api/person")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(request))
    ).andExpectAll(
        status().isBadRequest()
    );
}
```

Kode : Model Attribute Valid

```
@PostMapping(path = @"/person", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
@ResponseBody
@ResponseStatus(HttpStatus.OK)
public String createPerson(@ModelAttribute @Valid CreatePersonRequest request) {

    System.out.println(request.getHobbies());
    for (CreateSocialMediaRequest socialMedia : request.getSocialMedias()) {
        System.out.println(socialMedia.getName() + " : " + socialMedia.getLocation());
    }

    return new StringBuilder().append("Success create person ")
        .append(request.getFirstName()).append(" ")
        .append(request.getMiddleName()).append(" ")
        .append(request.getLastName())
}
```



Kode : Test Model Attribute Valid

```
@Test
void createPersonInvalid() throws Exception {
    mockMvc.perform(
        post("/person")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("middleName", "Kurniawan")
            .param("lastName", "Khannedy")
    ).andExpectAll(
        status().isBadRequest()
    );
}
```

Exception Handler

Exception Handler

- Saat terjadi error di Controller, seperti validation error, logic error, dan lain-lain
- Secara default, Spring akan mengembalikan response error sesuai jenis errornya
- Kadang, kita ingin membuat halaman atau response error sendiri
- Hal ini bisa kita buat dengan menggunakan @ControllerAdvice
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html>

Controller Advice

- Controller Advice adalah sebuah class yang dipanggil ketika sebuah jenis exception terjadi
- Dengan begitu kita bisa memanipulasi response yang akan dikembalikan ke user menggunakan Controller Advice ini



Kode : Error Controller

```
import org.springframework.web.bind.annotation.ControllerAdvice;  
  
@ControllerAdvice  
public class ErrorController {  
  
}
```

Exception Handler

- Setelah membuat Controller Advice, untuk menangkap exception dan mengubah response nya, kita perlu membuat Method seperti di Controller
- Namun kita tidak menggunakan annotation @RequestMapping, melainkan @ExceptionHandler
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html>
- Kita harus tentukan jenis exception apa yang akan ditangkap, dan jika butuh data exception nya, kita bisa tambahkan sebagai parameter di Method nya



Kode : Validation Exception Handler

```
@ControllerAdvice
public class ErrorController {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<String> methodArgumentNotValidException(MethodArgumentNotValidException exception) {
        return new ResponseEntity<>("Validation Error : " + exception.getMessage(), HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(ConstraintViolationException.class)
    public ResponseEntity<String> constraintViolationException(ConstraintViolationException exception) {
        return new ResponseEntity<>("Validation Error : " + exception.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```



Kode : Test Person API Validation

```
void createPersonValidationError() throws Exception {
    CreatePersonRequest request = new CreatePersonRequest();
    request.setMiddleName("Kurniawan");
    request.setHobbies(List.of("Coding", "Reading"));
    request.setSocialMedias(new ArrayList<>());
    request.getSocialMedias().add(new CreateSocialMediaRequest("Facebook", "facebook.com/ProgrammerZamanNow"));

    mockMvc.perform(
        post("/api/person")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .content(objectMapper.writeValueAsString(request))
    ).andExpectAll(
        status().isBadRequest(),
        content().string(Matchers.containsString("Validation Error :"))
    );
}
```

Error Page



Error Page

- Saat terjadi exception yang tidak tertangani oleh Exception Handler, secara default Spring WebMVC akan mengirim detail errornya ke path /error
- Jika tidak ada Controller Method dengan Route /error, maka Spring akan menampilkan default Page untuk error tersebut



Default Error Page

← → C ⌂

ⓘ localhost:8080/halaman-kosong

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon May 01 21:59:54 WIB 2023

There was an unexpected error (type=Not Found, status=404).



Error Page Properties

- Secara default, detail error tidak ditampilkan di error page, hal ini agar stacktrace tidak terexpose ketika terjadi error
- Semua detail properties untuk error page bisa kita setting di application properties dengan prefix server.error
- <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.server>

Membuat Error Detail Sendiri

- Kita juga bisa membuat error detail page sendiri jika mau, namun kita harus mematikan fitur error detail page bawaan dari Spring Boot dengan menambahkan properties
- **server.server.error.whitelabel.enabled=false**
- Selanjutnya kita bisa membuat controller dengan route /error, namun kita wajib mengimplement interface ErrorController
- <https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/servlet/error/ErrorController.html>
- Dan ketika kita membuat error page, jika ingin mendapatkan detail errornya, kita bisa menggunakan HttpServletRequest, dengan mengambil attribute dengan key prefix RequestDispatcher.ERROR_*



Kode : Error Page Controller

```
@Controller
public class ErrorPageController implements ErrorController {

    @RequestMapping(path = @"/error")
    public ResponseEntity<String> error(HttpServletRequest request) {
        Integer status = (Integer) request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
        String message = (String) request.getAttribute(RequestDispatcher.ERROR_MESSAGE);

        String html = """
            <html>
                <body>
                    <h1>$status - $message</h1>
                </body>
            </html>
        """.replace("$status", status.toString()).replace("$message", message);
        return ResponseEntity.status(status).body(html);
    }
}
```

Binding Result

Binding Result

- Secara default, jika terjadi error di @ModelAttribute atau @RequestBody, maka akan throw exception MethodArgumentNotValidException
- Kadang kita tidak ingin hal itu terjadi, misal kita ingin tetap masuk ke Controller Method, karena di dalam nya kita ingin menampilkan halaman errornya misalnya
- Pada kasus seperti itu, kita bisa tambahkan parameter BindingResult di sebelah parameter nya, secara otomatis detail error akan dimasukkan ke object BindingResult
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/BindingResult.html>



Kode : Person Controller

```
@PostMapping(path = @"/person", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public ResponseEntity<String> createPerson(@ModelAttribute @Valid CreatePersonRequest request,
                                             BindingResult bindingResult) {

    if (!bindingResult.getAllErrors().isEmpty()) {
        return ResponseEntity.badRequest().body("You send invalid data");
    }

    System.out.println(request.getHobbies());
    for (CreateSocialMediaRequest socialMedia : request.getSocialMedias()) {
        System.out.println(socialMedia.getName() + " : " + socialMedia.getLocation());
    }
}
```



Kode : Test Person Controller

```
@Test
void createPersonInvalid() throws Exception {
    mockMvc.perform(
        post("/person")
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)
            .param("middleName", "Kurniawan")
            .param("lastName", "Khannedy")
    ).andExpectAll(
        status().isBadRequest(),
        content().string(Matchers.containsString("You send invalid data"))
    );
}
```

Session Attribute

Session Attribute

- Seperti yang pernah dibahas di materi Java Servlet, bahwa di Java Servlet, kita bisa membuat Session
- Kita tidak akan membahas bagaimana cara melakukan management session di sini, karena sudah dibahas lengkap di kelas Java Servlet
- Spring WebMVC menyediakan cara mudah untuk mengakses data di Session menggunakan annotation @SessionAttribute
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/SessionAttribute.html>



Kode : User Class

```
no usages
7 ✓ @Data
8   @AllArgsConstructor
9   @NoArgsConstructor
0 public class User {
1
2     private String username;
3 }
4 |
```



Kode : Auth Controller

```
public class AuthController {  
  
    @PostMapping(path = @"/auth/login", consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)  
    public ResponseEntity<String> login(@RequestParam(name = "username") String username,  
                                         @RequestParam(name = "password") String password,  
                                         HttpServletRequest servletRequest,  
                                         HttpServletResponse servletResponse) {  
  
        if ("eko".equals(username) && "nahasia".equals(password)) {  
            HttpSession session = servletRequest.getSession(true);  
            session.setAttribute("user", new User(username));  
  
            Cookie cookie = new Cookie("username", username);  
            cookie.setPath("/");  
            servletResponse.addCookie(cookie);  
  
            return new ResponseEntity<>("OK", HttpStatus.OK);  
        } else {  
            return new ResponseEntity<>("Unauthorized", HttpStatus.UNAUTHORIZED);  
        }  
    }  
}
```



Kode : User Controller

```
@Controller
public class UserController {

    @GetMapping(path = "/user/current")
    @ResponseBody
    public String getUser(@SessionAttribute(name = "user") User user) {
        return "Hello " + user.getUsername();
    }
}
```



Kode : Test User Controller

```
└── @SpringBootTest
    ├── @AutoConfigureMockMvc
    └── class UserControllerTest {
        └── @Autowired
            private MockMvc mockMvc;
        └── @Test
            void getUser() throws Exception {
                mockMvc.perform(
                    get("/user/current")
                        .sessionAttr("user", new User("eko"))
                ).andExpectAll(
                    status().isOk(),
                    content().string(Matchers.containsString("Hello eko"))
                );
            }
    }
}
```

MVC Config



MVC Config

- Saat membuat aplikasi web menggunakan Spring Web MVC, kita bisa menambahkan pengaturan untuk Spring Web MVC
- Caranya kita perlu membuat sebuah Bean configuration turunan dari WebMvcConfigurer
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/config/annotation/WebMvcConfigurer.html>
- Ada banyak sekali method yang bisa kita override untuk menambah konfigurasi yang ada di Spring Web MVC



Kode : MyWebConfig

```
✓ import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class MyWebConfig implements WebMvcConfigurer {

}
```

Interceptor



Interceptor

- Saat kita belajar Java Servlet, kita tahu ada fitur yang bernama WebFilter, yang tugasnya mirip sebagai middleware
- Di Spring WebMVC, kita bisa menggunakan fitur bernama Interceptor, untuk melakukan hal yang sama
- Cara melakukan registrasi Interceptor adalah dengan membuat class turunan dari HandlerInterceptor, lalu menambahkan menggunakan InterceptorRegistry di method addInterceptors() WebMvcConfigurer
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html>



Kode : Session Interceptor

```
@Component
public class SessionInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        HttpSession session = request.getSession(true);
        User user = (User) session.getAttribute("user");
        if (user == null) {
            response.sendRedirect("/login");
            return false;
        }
        return true;
    }
}
```



Kode : Registrasi Session Interceptor

```
@Configuration
public class MyWebConfig implements WebMvcConfigurer {

    @Autowired
    private SessionInterceptor sessionInterceptor;

    2 usages
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(sessionInterceptor).addPathPatterns("/user/*");
    }
}
```



Kode : Test User Controller

```
@Test
void getUserInvalid() throws Exception {
    mockMvc.perform(
        get("/user/current")
    ).andExpectAll(
        status().is3xxRedirection()
    );
}
```

Ant Path Matcher

- Spring kebanyakan menggunakan Ant Path Matcher untuk pattern penulisan path
- Format Ini diambil dari sebuah library bernama Apache Ant
- Untuk lebih detail, kita bisa cek di class AntPathMatcher
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html>

Argument Resolver

Argument Resolver

- Saat kita menambahkan sebuah parameter di Controller Method, maka Spring Web MVC akan mencoba mencari dari mana data tersebut berasal
- Oleh karena itu kita perlu tambahkan penanda seperti ModelAttribute, RequestBody, RequestParam, dan lain-lain
- Kita juga bisa membuat sebuah ArgumentResolver, yaitu class yang digunakan untuk mengisi object argument yang kita inginkan secara otomatis
- Spring akan otomatis memanggil ArgumentResolver tersebut, ketika terdapat parameter dengan tipe data yang sudah kita tentukan



Handler Method Argument Resolver

- Untuk membuat Argument Resolver, kita harus membuat class turunan HandlerMethodArgumentResolver
- Setelah itu kita harus registrasikan ke WebMvcConfigurer melalui method addArgumentResolvers()
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/method/support/HandlerMethodArgumentResolver.html>



Kode : Partner Class

```
no usages
7 ✓ @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class Partner {
11
12     private String id;
13
14     private String name;
15 }
16 |
```



Kode : Partner Argument Resolver

```
@Component
public class PartnerArgumentResolver implements HandlerMethodArgumentResolver {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return parameter.getParameterType().equals(Partner.class);
    }

    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) {
        HttpServletRequest servletRequest = (HttpServletRequest) webRequest.getNativeRequest();
        String apiKey = servletRequest.getHeader("X-API-KEY");
        if (apiKey != null) {
            return new Partner(apiKey, "Sample Partner");
        }
        throw new RuntimeException("Unauthorized Exception");
    }
}
```

Kode : Web Mvc Configurer

```
@Configuration
public class MyWebConfig implements WebMvcConfigurer {

    @Autowired
    private SessionInterceptor sessionInterceptor;

    @Autowired
    private PartnerArgumentResolver partnerArgumentResolver;

    2 usages
    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {
        💡     resolvers.add(partnerArgumentResolver);
    }
}
```



Kode : Partner Controller

```
@Controller
public class PartnerController {

    @GetMapping(path = @v"/partner/current")
    @ResponseBody
    @ResponseStatus(HttpStatus.OK)
    public String getPartner(Partner partner) {
        return partner.getId() + ":" + partner.getName();
    }
}
```



Kode : Test Partner Controller

```
@Test
void getPartner() throws Exception {
    mockMvc.perform(
        get("/partner/current")
            .header("X-API-KEY", "SAMPLE")
    ).andExpectAll(
        status().isOk(),
        content().string(containsString("SAMPLE:Sample Partner"))
    );
}
```

Kekurangan Menggunakan Argument Resolver

- Argument Resolver hanyalah untuk mengisi data yang terdapat di parameter Controller Method
- Kita tidak bisa memodifikasi Http Response seperti di Interceptor
- Oleh karena itu, jika butuh melakukan modifikasi, kita bisa kombinasikan Interceptor dan Argument Resolver
- Misal dengan mengirim data dari Interceptor melalui Request Attribute, dan diterima di Argument Resolver

Static Resource

Static Resource

- Saat kita membuat website, kita sering sekali membuat konten static, misal html, css, javascript, image, video, dan sejenisnya
- Jika kita handle semua dengan membuat Controller atau Servlet, maka akan menyulitkan
- Untungnya Spring WebMVC, memiliki fitur untuk menangani Static Resource ini
- Kita bisa menambahkan semua resource static di folder static di directory resources
- Ketika kita mengakses Path di Spring Web MVC, pertama Spring akan mencoba mencari Controller yang memiliki Request Mapping tersebut, jika tidak ada, secara otomatis akan mencoba mengakses Static Resource, jika ternyata masih tidak ada, maka barus akan mengembalikan 404 Not Found



Kode : /resources/static/index.html

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport"
6         content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
7     <meta http-equiv="X-UA-Compatible" content="ie=edge">
8     <title>Hello Static</title>
9 </head>
10 <body>
11     <h1>Hello Static</h1>
12 </body>
13 </html>
14
```



Kode : Unit Test Static Resource

```
@Test
void getStaticResource() throws Exception {
    mockMvc.perform(
        get("/index.html")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Hello Static"))
    );
}
```

View

View

- Sampai sekarang, kita sudah membahas tentang Controller di Spring WebMVC
- Kita belum pernah membahas tentang View
- Sebelumnya kita hanya mengembalikan koten web secara manual di Controller
- Spring WebMVC sendiri tidak membuat fitur untuk View / Templating secara manual
- Spring WebMVC mengintegrasikan banyak sekali teknologi untuk Templating yang digunakan sebagai bagian dari View nya

View yang Didukung

- Spring WebMVC mendukung banyak sekali library untuk View, misalnya
- JSP (Java Server Page)
- Apache Velocity : <https://velocity.apache.org/>
- Apache Freemarker : <https://freemarker.apache.org/>
- Mustache : <https://mustache.github.io/>
- Thymeleaf : <https://www.thymeleaf.org/>
- Di video ini, kita akan bahas integrasi dengan Mustache, namun untuk detail dari Mustache tidak dibahas di materi ini, karena sudah dibahas di materi kelas Java Mustache



Mustache

- Saat kita menambahkan dependecy Spring Boot Mustache, kita tidak perlu melakukan pengaturan secara manual lagi untuk membuat Mustache
- Semua sudah diatur secara otomatis oleh Spring Boot
- Template Mustache bisa kita simpan di folder /resources/templates/ dengan extension .mustache
- Semua pengaturan bisa digunakan via application properties
- <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties.templates>

Model And View



Model And View

- Untuk menampilkan View, kita bisa mengembalikan return object ModelAndView pada Controller Method
- Dalam ModelAndView, kita bisa memasukkan data template yang dipilih untuk View, dan juga Model yang akan ditampilkan di View
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/ModelAndView.html>



Kode : Hello Template

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport"
6     content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
7   <meta http-equiv="X-UA-Compatible" content="ie=edge">
8   <title>{{title}}</title>
9 </head>
10 <body>
11   <h1>Hello {{name}}</h1>
12 </body>
13 </html>
```



Kode : Hello Controller

```
@GetMapping(path = @v"/web/hello")
public ModelAndView hello(@RequestParam(name = "name", required = false) String name) {
    return new ModelAndView("hello", Map.of(
        "title", "Belajar View",
        "name", name
    ));
}
```



Kode : Test Hello Controller

```
@Test
void helloView() throws Exception {
    mockMvc.perform(
        get("/web/hello").queryParam("name", "Eko")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Belajar View")),
        content().string(Matchers.containsString("Hello Eko"))
    );
}
```

Redirect

Redirect

- Jika kita ingin melakukan Redirect, selain menggunakan HttpServletResponse
- Jika pada kasus di Controller Method mengembalikan ModelAndView
- Kita bisa gunakan view name dengan prefix redirect:
- Secara otomatis Spring Web MVC akan melakukan redirect

Kode : Hello Controller

```
@GetMapping(path = "/web/hello")
public ModelAndView hello(@RequestParam(name = "name", required = false) String name) {
    if (Objects.isNull(name)) {
        return new ModelAndView("redirect:/web/hello?name=Guest");
    }
    return new ModelAndView("hello", Map.of(
        "title", "Belajar View",
        "name", name
    ));
}
```



Kode : Test Redirect

```
@Test
void helloViewRedirect() throws Exception {
    mockMvc.perform(
        get("/web/hello")
    ).andExpectAll(
        status().is3xxRedirection()
    );
}
```

Rest Controller



Rest Controller

- Sebelumnya kita sudah tahu untuk membuat Controller, kita menggunakan annotation Controller
- Spring Web MVC menyediakan annotation khusus untuk membuat Controller khusus untuk RESTful API, yaitu annotation RestController
- RestController ini sebenarnya gabungan antara @Controller dan @ResponseBody, yang artinya secara otomatis semua return Controller Method tersebut dianggap sebagai Response Body



Kode : Todo Controller

```
@RestController
public class TodoController {

    3 usages

    private List<String> todos = new ArrayList<>();

    @PostMapping(path = @v"/todos", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<String> addTodo(@RequestParam("todo") String todo) {
        todos.add(todo);
        return todos;
    }

    @GetMapping(path = @v"/todos", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<String> getTodos(@RequestParam("todo") String todo) {
        return todos;
    }
}
```

Kode : Test Todo Controller

```
@Test
void addTodo() throws Exception {
    mockMvc.perform(
        post("/todos")
            .accept(MediaType.APPLICATION_JSON)
            .param("todo", "Eko")
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Eko"))
    );
}
```

```
@Test
void getTodos() throws Exception {
    mockMvc.perform(
        get("/todos")
            .accept(MediaType.APPLICATION_JSON)
    ).andExpectAll(
        status().isOk(),
        content().string(Matchers.containsString("Eko"))
    );
}
```

Rest Template



Rest Template

- Saat kita membuat aplikasi Web / RESTful API, kadang kita juga butuh memanggil/mengirim data ke server Web/RESTful API lainnya
- Spring sudah menyediakan class bernama RestTemplate, yang bisa kita gunakan sebagai HTTP Client / RESTful Client
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>



Rest Template Builder

- Untuk membuat RestTemplate, kita bisa menggunakan RestTemplateBuilder, yang secara otomatis sudah dibuatkan sebagai Bean oleh Spring Boot
- Sebelum membuat RestTemplate, kita bisa melakukan konfigurasi terlebih dahulu di RestTemplateBuilder
- <https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/client/RestTemplateBuilder.html>



Kode : Membuat Rest Template

```
@SpringBootApplication
public class BelajarSpringWebmvcApplication {

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder
            .setConnectTimeout(Duration.ofSeconds(2L))
            .setReadTimeout(Duration.ofSeconds(2L))
            .build();
    }
}
```

Request & Response Entity

- Untuk mengirim request, kita bisa menggunakan RestTemplate.exchange(), dimana kita perlu membuat RequestEntity
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/RequestEntity.html>
- Response dari RestTemplate adalah object ResponseEntity
- <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/ResponseEntity.html>
- Pada kasus server mengembalikan data JSON, kita bisa otomatis melakukan konversi menjadi Object dengan bantuan Jackson secara otomatis



Kode : Test Rest Template Add Todo

```
@Autowired
private RestTemplate restTemplate;

@Test
void addTodo() {
    String url = "http://localhost:" + port + "/todos";

    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(List.of(MediaType.APPLICATION_JSON));

    MultiValueMap<String, Object> form = new LinkedMultiValueMap<>();
    form.add("todo", "Belajar Spring WebMVC");

    RequestEntity<MultiValueMap<String, Object>> request = new RequestEntity<>(form, headers, HttpMethod.POST, URI.create(url));

    ResponseEntity<List<String>> response = restTemplate.exchange(request, new ParameterizedTypeReference<>() {});
    Assertions.assertEquals(HttpStatus.OK, response.getStatusCode());
    Assertions.assertTrue(response.getBody().contains("Belajar Spring WebMVC"));
}
```



Kode : Test Rest Template Get Todos

```
@Test
void getTodos() {
    String url = "http://localhost:" + port + "/todos";

    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(List.of(MediaType.APPLICATION_JSON));

    RequestEntity<MultiValueMap<String, Object>> request = new RequestEntity<>(headers, HttpMethod.GET, URI.create(url));

    ResponseEntity<List<String>> response = restTemplate.exchange(request, new ParameterizedTypeReference<>() {});
    Assertions.assertEquals(HttpStatus.OK, response.getStatusCode());
    Assertions.assertTrue(response.getBody().contains("Belajar Spring WebMVC"));
}
```

Servlet Integration



Servlet Integration

- Saat kita membuat WebServlet atau WebFilter, secara default Spring WebMVC tidak akan meregistrasikannya ke Embedded Apache Tomcat
- Jika kita ingin membuat WebServlet dan WebFilter, dan ingin Spring otomatis meregistrasikannya ke Embedded Apache Tomcat, maka kita perlu menggunakan annotation `ServletComponentScan`
- <https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/servlet/ServletComponentScan.html>



Kode : Hello Servlet

```
@WebServlet(urlPatterns = "/servlet/hello")
public class HelloServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().println("Hello from Servlet");
    }
}
```



Kode : Test Hello Servlet

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

    @LocalServerPort
    private Integer port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void helloServlet() {
        String response = restTemplate.getForObject("http://localhost:" + port + "/servlet/hello", String.class);
        Assertions.assertEquals("Hello from Servlet", response.trim());
    }
}
```