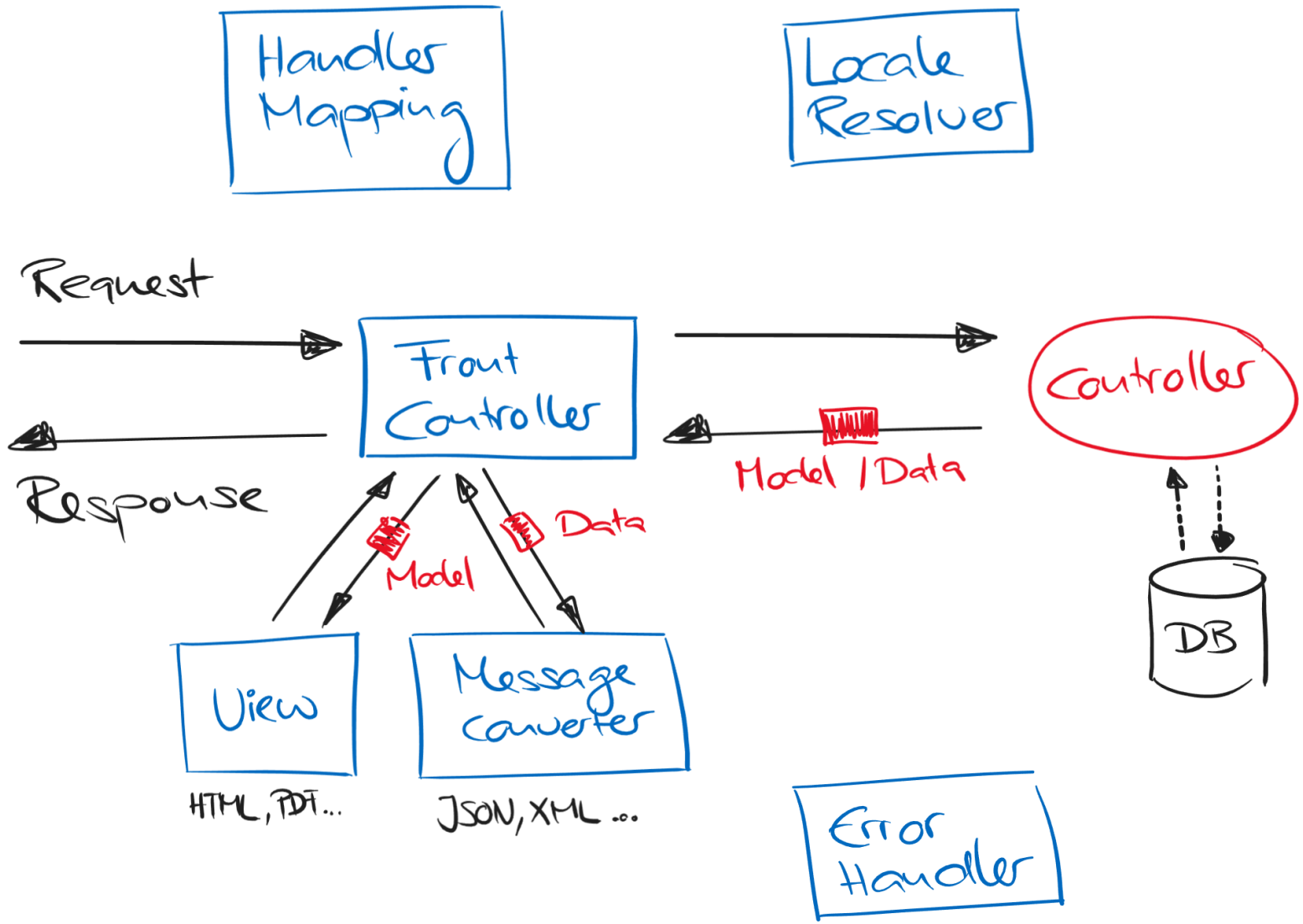
Restful Web Services With Spring Boot

Spring MVC

What is Spring MVC?

What is Spring MVC?

- Request based web framework
- Implements Model View Controller (MVC) pattern
- FrontController a.k.a. `DispatcherServlet` delegates requests
- Extended with MessageConverters to create RESTful web services


Handler Mapping
Locale Resolver
Request
Front Controller
Controller
Response
Model / Data
Model
Data
View
Message converter
DB
HTML, PDF...
JSON, XML ...
Error Handler

Spring MVC Setup

Add Dependencies

- Adding Spring MVC Starter to your project and getting dependencies for Spring MVC, Jackson, and Tomcat
- Thymeleaf Starter provides a templating engine to render HTML pages

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

Setup

- Starters configure infrastructre Spring beans automatically
  - Thymeleaf
    - Templateing engine
    - Caching enabled -> Production Defaults
  - Spring MVC
    - `ContextLoaderListener`
    - `DispatcherServlet`
    - `@EnableWebMvc` with formatters, converters and validators
    - Static resources served from `/static`, `/public`, `/resources` or `/META-INF/resources`
    - Templates served from `/templates`
    - Provides default `/error` mapping
    - Default `MessageSource` for I18N

Controllers

Simple Controller with View

```
@Controller
public class ApplicationNameController {

  @Value("${spring.application.name}")
  private String appName;

  @GetMapping("/")
  public String homePage(Model model) {
    model.addAttribute("appName", appName);
    return "home";
  }
}
```

Thymeleaf Template

- HTML template added to `/templates`
- Static resources can be served from `/static`, `/public`, `/resources` or `/META-INF/resources`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Home Page</title>
</head>
<body>
  <h1>Hello !</h1>
  <p>
    Welcome to <span th:text="${appName}">Our App</span>
  </p>
</body>
</html>
```

Request Mappings

@RequestMapping

Generic

- `@RequestMapping(path = "…" , method = {…})`

HTTP default methods

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

Status Codes & Media Types

HTTP Status Codes

- Status codes are an indicator of the result of the server's attempt to satisfy the request
- Divided in categories
  - 1XX: Informational
  - 2XX: Success
  - 3XX: Redirection
  - 4XX: Client Error
  - 5XX: Server Error

Media Types

- Accept & Content-Type HTTP Headers
  - Client and server describes the content

HttpMessageConverter

- Converts HTTP request/response body data
  - XML: JAXP Source*, JAXB2 mapped object*, Jackson-Dataformat-XML*
  - GSON*, Jackson JSON*
  - Feed data* such as Atom/RSS
  - Google protocol buffers*
  - Form-based data
  - `Byte[]`, `String`, `BufferedImage`
- Converters enabled automatic with Spring Boot AutoConfiguration
  - Don't use `@EnableWebMvc` (different behavior)
  - Define explicitly with `WebMvcConfigurer`

* Third party libraries required on classpath

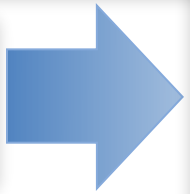
Rest Controller

@ResponseBody

- Converter for response used because of `@ResponseBody`
- Converter handles rendering, no view involved
- Uses `Accept-Header` for Content Negotiation

```
@Controller
public class CustomerController {
  @GetMapping(path="/customers/{id}")
  public @ResponseBody Customer showCustomer(
      @PathVariable("id") long id) {
        // …
    }
  }
```

```
GET /customers/42
Host: www.example.com
Accept: application/json
```



```
HTTP/1.1 200 OK
Date: …
Content-Length: 723
Content-Type: application/json
{
  "customer": {
    "id": 42,
    "address": [ … ],
    …
  }
}
```

@RestController Simplification

```
@Controller
public class CustomerController {

  @GetMapping(path="/customers/{id}")
  public @ResponseBody Customer showCustomer(@PathVariable("id") long id) {
    // …
  }
}
```

```
@RestController
public class CustomerController {

  @GetMapping(path="/customers/{id}")
  public Customer showCustomer(@PathVariable("id") long id) {
    // …
  }
}
```

`@ResponseStatus`

- Controller returns by default status `200 OK`
- `@ResponseStatus` to override status code
- `@ResponseStatus` also on void methods

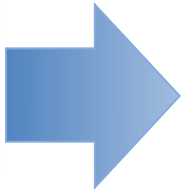
```
@RestController
public class CustomerController {

  @PutMapping(path="/customers/{id}")
  @ResponseStatus(HttpStatus.NO_CONTENT)
  public void updateCustomer(@PathVariable("id") long id, @RequestBody Customer customer) {
    // Update customer data
  }
}
```

RESTful Services with Spring MVC

REST Blueprint

Retrieving a Representation: GET

```
GET /customers/42
Host: www.example.com
Accept: application/xml,…
…
```

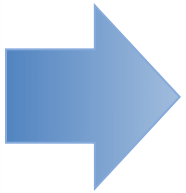


```
HTTP/1.1 200OK
Date: …
Content-Length: 1456
Content-Type: application/xml
<customer id="42">
  …
</customer>
```

```
@GetMapping(path="/customers/{id}")
public @ResponseBody Customer getCustomer(@PathVariable("id") long id) {
  return customerService.findCustomerById(id);
}

@RequestMapping(path="/customers/{id}", method=RequestMethod.GET)
```

Creating a New Resource: POST (1)

```
POST /customers/42/addresses
Host: www.example.com
Content-Type: application/xml
<item>
…
</item>
```



```
HTTP/1.1 201 Created
Date: …
Content-Length: 0
Location: https://{…}/42/addresses/abc
```

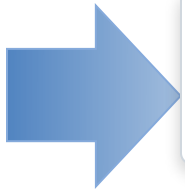
- The most complicated to implement
  - `201 Created` requires the `Location` header for the new resource
- `ServletUriComponentsBuilder.fromCurrentRequestUri()`
  - Returns a `UriComponentsBuilder` initialized to URL of current Controller method
  - Useful as our new resource is a sub-path of the `POST` URL

Creating a New Resource: POST (2)

```
@PostMapping("/customers/{id}/addresses")
public ResponseEntity<Void> createAddress(
  @PathVariable long id,
  @RequestBody Address newAddress) {

  // Add the new address to the customer
  customerService.findCustomerById(id).addAddress(newAddress);

  // Build the location URI of the new address
  URI location = ServletUriComponentsBuilder
       .fromCurrentRequestUri()
       .path("/{addressId}")
       .buildAndExpand(newAddress.getId())
       .toUri();

  // Explicitly create a 201 Created response
  return ResponseEntity.created(location).build();
}

@RequestMapping(path="/customers/…", method=RequestMethod.POST)
```

Updating a Resource: PUT

```
PUT /customers/42/addresses/abc
Host: www.example.com
Content-Type: application/xml
<item>
…
</item>
```

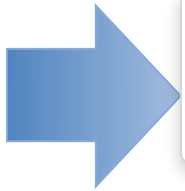


```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```
@PutMapping(path="/customers/{customerId}/addresses/{addressId}")
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateAddress(
    @PathVariable("customerId") long customerId,
    @PathVariable("addressId") String addressId,
    @RequestBody Address item) {
  customerService.findCustomerById(customerId).updateAddress(addressId, item));
}

@RequestMapping(path="/customers/…", method=RequestMethod.PUT)
```

Deleting a Resources: DELETE

```
DELETE /customers/123/addresses/abc
Host: www.example.com
…
```



```
HTTP/1.1 204 No Content
Date: …
Content-Length: 0
```

```
@DeleteMapping(path="/customers/{customerId}/addresses/{addressId}")
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void deleteAddress(
    @PathVariable("customerId") long customerId,
    @PathVariable("addressId") String addressId) {
  customerService.findCustomerById(customerId).deleteAddress(addressId);
}

@RequestMapping(path="/customers/…", method=RequestMethod.DELETE)
```

More Annotations & Automatic Conversion

URL Examples

```
@GetMapping("/accounts")
public String show(HttpServletRequest request, Model model)


@GetMapping("/customers/{id}/addresses/{addressId}")
public String show(@PathVariable("id") Long customerId,
    @PathVariable int addressId,
    Model model,
    Locale locale,
    @RequestHeader("user-agent") String agent)


@GetMapping("/customers")
public String show(@RequestParam Long customerId,
    @RequestParam("item") int addressId,
    Principal user,
    Map<String,Object> model,
    HttpSession session,
    @CookieValue("language") String lang)
```

Exceptions

`@ResponseStatus` & Exceptions

- Can also annotate exception classes with this
- Given status code used when an unhandled exception is thrown from *any* controller method

```
@ResponseStatus(HttpStatus.NOT_FOUND)  // 404
public class CustomerNotFoundException extends RuntimeException {
  // …
}
```

```
@GetMapping(value="/customers/{id}")
public Customer showCustomer(@PathVariable("id") long id, Model model) {
   Customer customer = customerService.findCustomerById(id);
   if (customer == null) throw new CustomerNotFoundException(id);
   return customer;
}
```

`@ExceptionHandler` & `ControllerAdvice`

- `@ExceptionHandler` method on controller handles Exception
- Supports `@ResponseStatus` and custom error message
- `@RestControllerAdvice` and `@ControllerAdvice` makes them available for all controllers

```
@Slf4j
@ControllerAdvice
public class RestCustomerControllerAdvice {

  @ExceptionHandler(CustomerNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  public void notFound(CustomerNotFoundException e){
    log.error("Error occured: {}", e);
  }
}
```

Problem Details (RFC 7807)

```java
@ControllerAdvice(annotations = RestController.class)
public class QuestionsControllerAdvice {

    @ExceptionHandler(UnknownAnswerException.class)
    public ProblemDetail handleUnknownAnswerException(UnknownAnswerException ex){
        ProblemDetail problemDetail = ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
        problemDetail.setType(URI.create("https://42talents.com/problems/answers"));
        problemDetail.setTitle("Unknown Answer.");
        problemDetail.setDetail(
          String.format("Answer for Question '%s' not found.", ex.getQuestion())
        );
        return problemDetail;
    }
}
```

https://www.rfc-editor.org/rfc/rfc7807

RestTemplate

RestTemplate

```
RestTemplate restTemplate = new RestTemplate();
```

HTTP	RESTTEMPLATE
DELETE	delete(String, String…)
GET	getForObject(String, Class, String…)
HEAD	headForHeaders(String, String…)
OPTIONS	optionsForAllow(String, String…)
POST	postForLocation(String, Object, String…)
PUT	put(String, Object, String…)

RestTemplate with Spring Boot

- `RestTemplateBuilder` is a `@Bean` registered AutoConfiguration
- When using special starters `Customizers` might run and adjust the `RestTemplateBuilder`
- Create your own instance with `RestTemplateBuilder`

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder restTemplateBuilder) {
  return new RestTemplateBuilder()
    .setConnectTimeout(Duration.ofSeconds(3))
    .setReadTimeout(Duration.ofSeconds(3))
    .build();
}
```

`RestTemplate` Usage Examples

```
RestTemplate template = … ;
String uri = "http://example.com/customers/{id}/addresses";

// GET all customer addresses for an existing customer with ID 1
CustomerAddress[] addresses = template.getForObject(uri, CustomerAddress[].class, 1);

// POST to create a new address
CustomerAddress address = … // create address object
URI addressLocation = template.postForLocation(uri, address, 1);

// PUT to update the address
address.setOrder(99);
template.put(addressLocation, address);

// DELETE to remove that item again
template.delete(addressLocation);
```

Using ResponseEntity

```
String uri = "http://example.com/customers/{id}";


ResponseEntity<Customer> response =
     restTemplate.getForEntity(uri, Customer.class, 1);


assert(response.getStatusCode().equals(HttpStatus.OK));


long modified = entity.getHeaders().getLastModified();


Customer customer = response.getBody();
```

RequestEntity and ResponseEntity

```
// POST with HTTP BASIC authentication
RequestEntity<CustomerAddress> request = RequestEntity
     .post(new URI(itemUrl))
     .getHeaders().add(HttpHeaders.AUTHORIZATION,
             "Basic " + getBase64EncodedCredentials())
     .accept(MediaType.APPLICATION_JSON)
     .body(newAddress);


ResponseEntity<Void> response =
         restTemplate.exchange(request, Void.class);
```

Spring HTTP Interface Client

Spring HTTP Interface Client

```
@HttpExchange(url = "/health", accept = MediaType.APPLICATION_JSON_VALUE)
public interface HttpHealthActuatorClient {

    @GetExchange
    HealthResponse getHealth();

    @GetExchange("/{component}")
    HealthResponse getHealth(@PathVariable String component);
}
```

https://docs.spring.io/spring-framework/docs/6.0.0/reference/html/integration.html#rest-http-interface

Important notes

Important notes

- Spring MVC is the base technology for frameworks like Spring Webflow and Spring Data REST, Springdoc OpenAPI, ...
- API Design Matters
  - URIs represent resources, not actions
  - HTTP verbs are general, but can be used many ways
- Easy testing
  - Out-of-container, Spring MockMVC

Spring Data Rest

Hypermedia & REST

Principles of REST

- *Resources* expose easily understood directory structure URIs
- *Representations* transfer JSON or XML to represent data objects and attributes
- *Messages* use HTTP methods explicitly (for example, GET, POST, PUT, and DELETE)
- *Stateless interactions* store no client context on the server between requests. State dependencies limit and restrict scalability. The client holds the session state.

See also: https://martinfowler.com/articles/richardsonMaturityModel.html

What is Hypermedia

- An important aspect of REST
- For building services that decouple client and server
- Allowing them to evolve independently
- Representations returned for REST resources contain data and links

What is Spring Data REST?

What is Spring Data REST?

- *Spring Data REST* exposes your Spring Data Repositories as REST Service
- REST Service built on top of *Spring HATEOAS*
- Uses *HAL* by default to render links

See also: https://spring.io/projects/spring-hateoas

See also: http://gtramontina.com/h-factors/

See also: http://amundsen.com/hypermedia/hfactor/

Spring Data REST Setup

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

What happens behind the scenes?

- Starter additionally configures infrastructure Spring beans automatically
  - Spring MVC
  - Spring HATEOAS
  - Jackson Marshaller

Entities & Repositories

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format("Customer[id=%d, firstName='%s', lastName='%s’]",id,firstName,lastName);
    }
}
```

JPA & Rest Repository

- All Repository automatically exposed as a REST Service
- `@RepositoryRestResource` used to override defaults

```
@RepositoryRestResource(path ="clients")
public interface CustomerRepository extends JpaRepository<Customer,Long> {

    List<Customer> findByLastName(@Param("lastName") String lastName);

    @Query("select e from Customer e where e.lastName = :lastName")
    List<Customer> findQueryByLastName(@Param("lastName") String name);
}
```

```
# application.properties
spring.data.rest.basePath=/api
```

Important notes

- You are exposing with Spring Data REST your whole Database
- Tightly coupled to the database, changing data structure becomes very difficult
- Links should be rendered depending on your business needs
- Use Spring MVC and Spring HATEOAS to add additional operations