



Spring + REST

Michael Inden

Freiberuflicher Consultant und Trainer

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael_inden@hotmail.com

Blog: <https://jaxenter.de/author/minden>

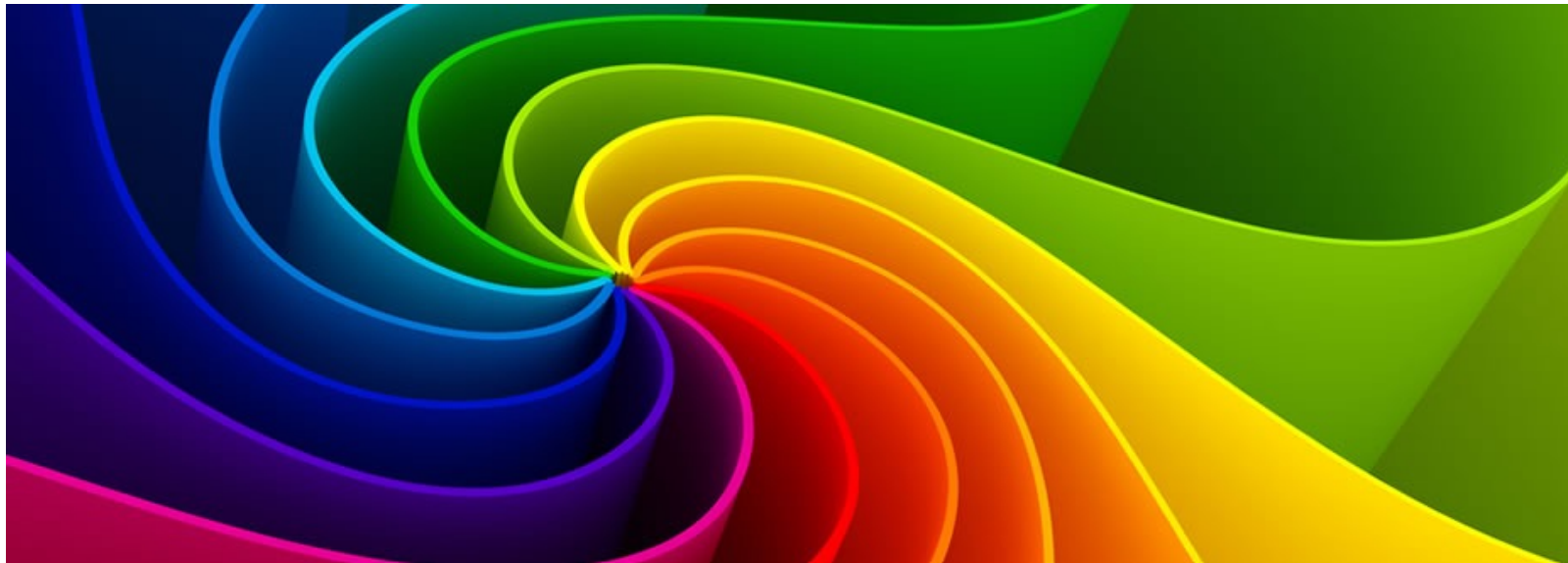
<https://www.wearedevelopers.com/magazine/java-records>

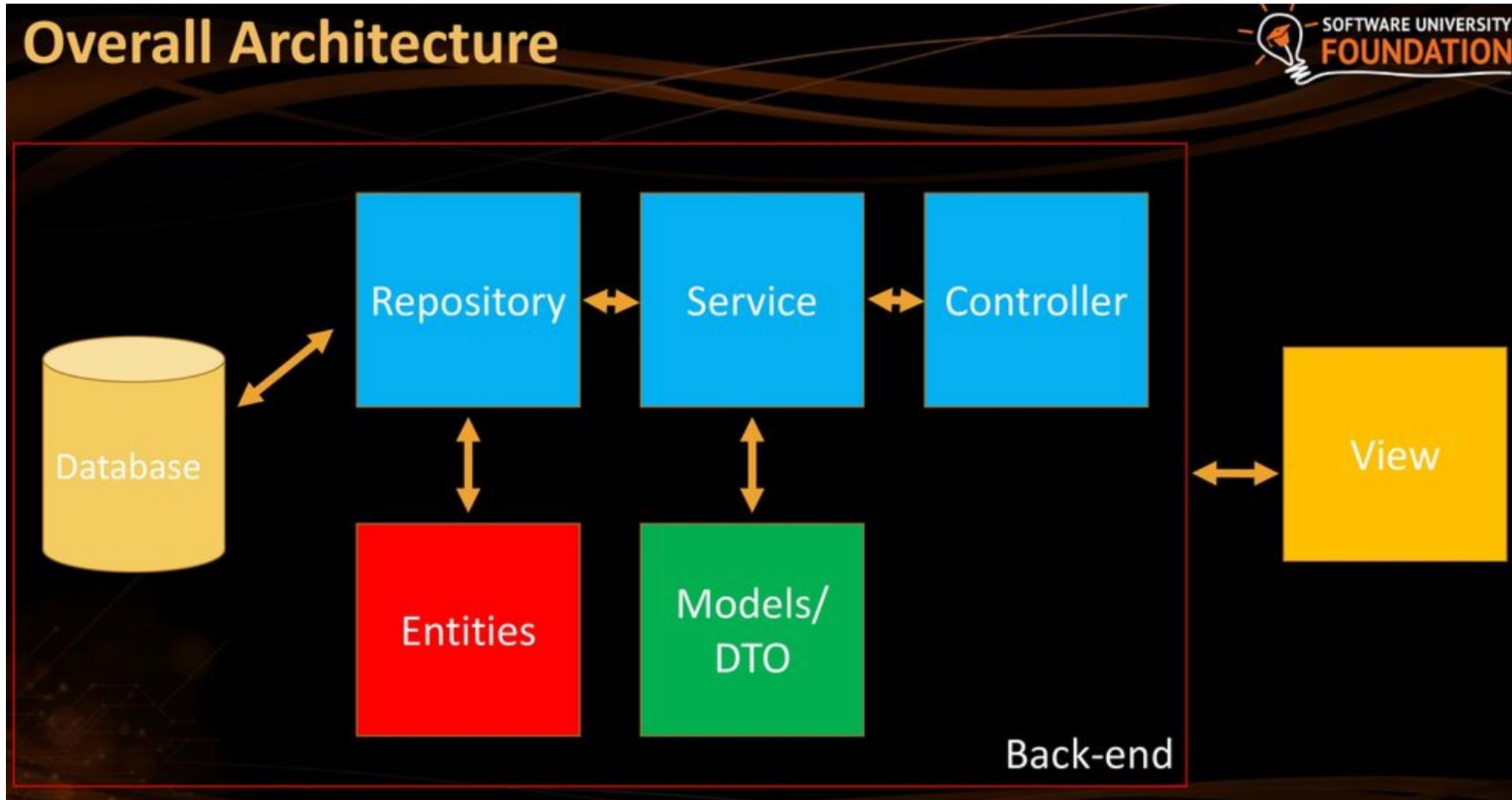
Kurse: **Bitte spricht mich an!**



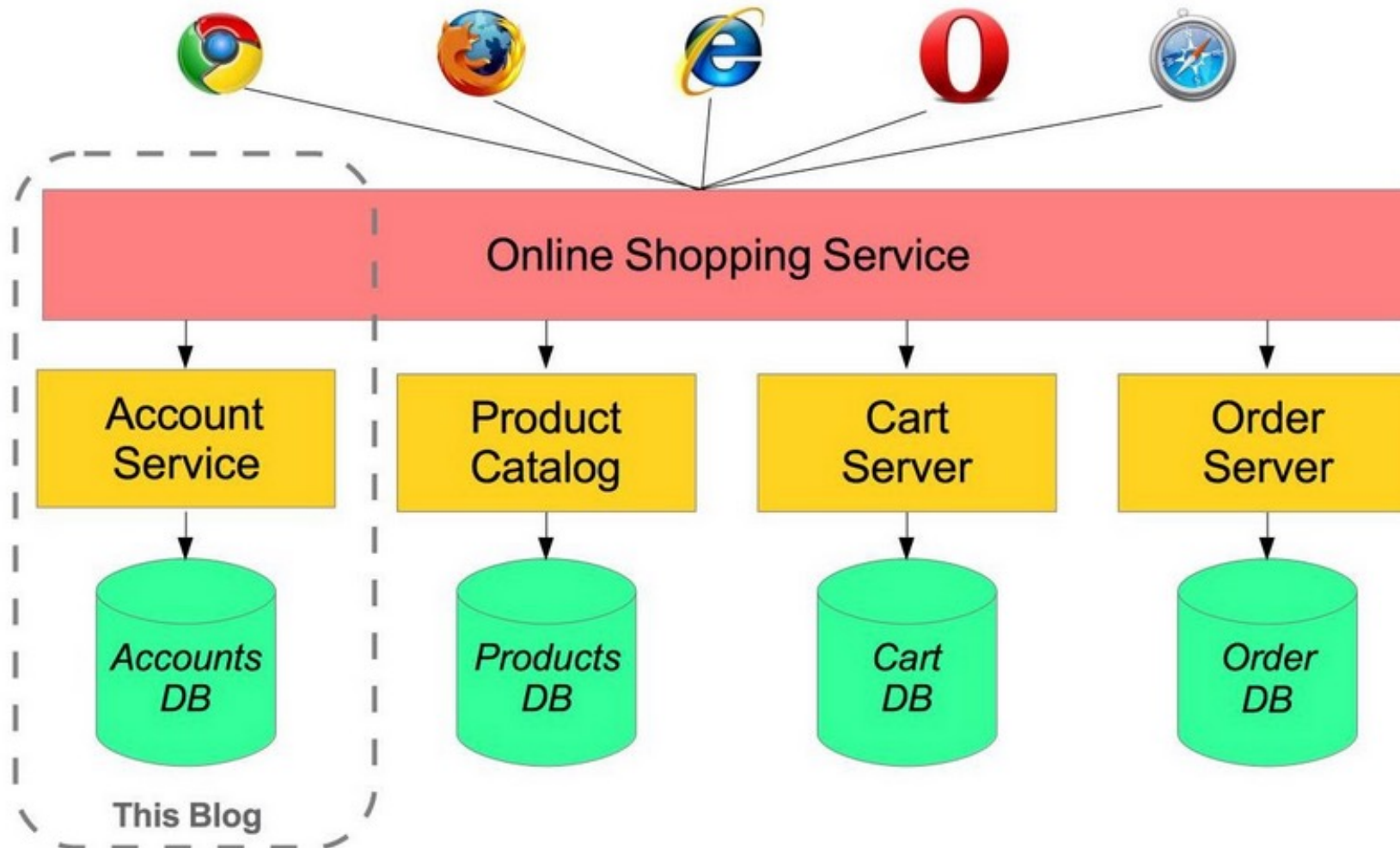


Architektur in Spring

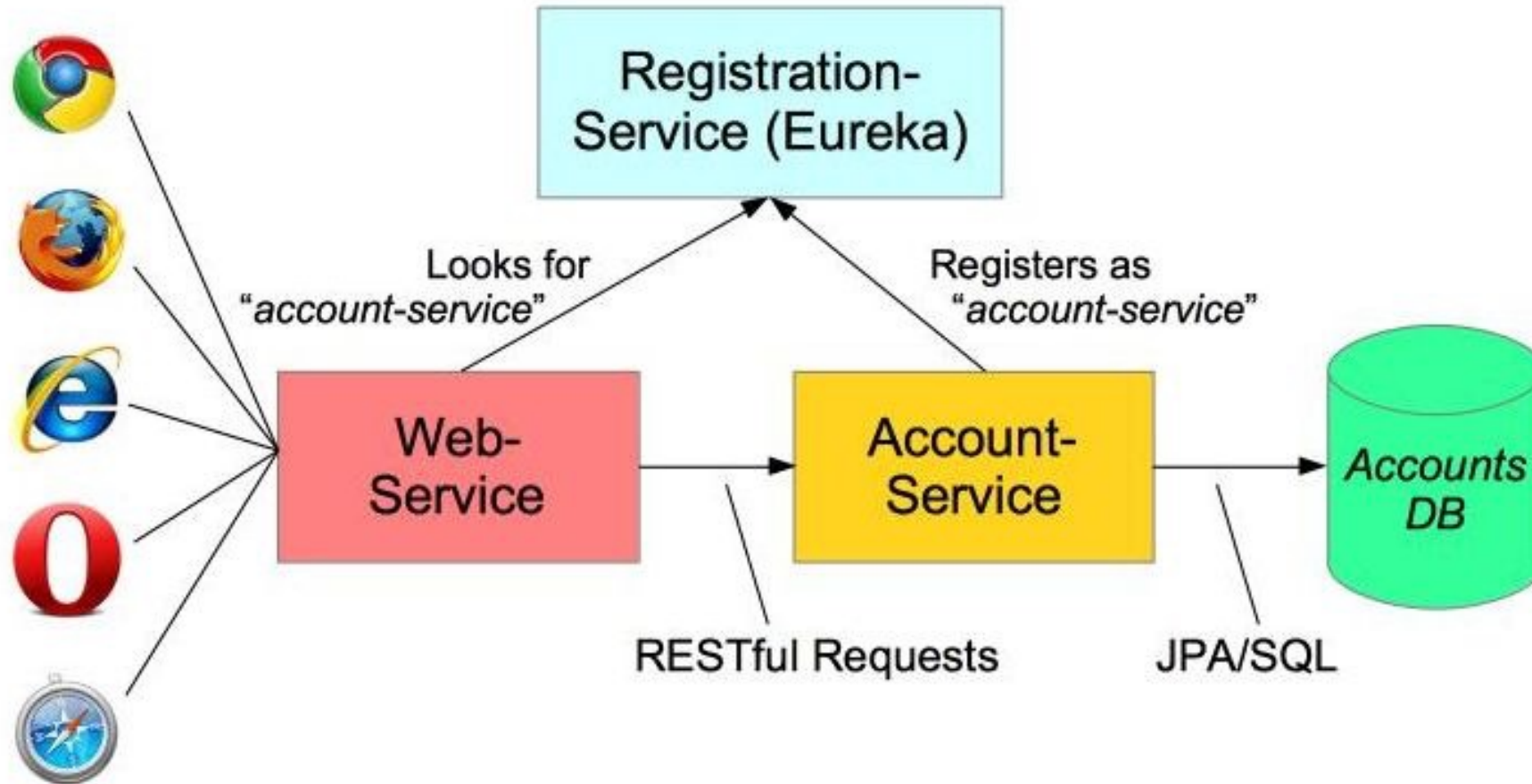




Microservice

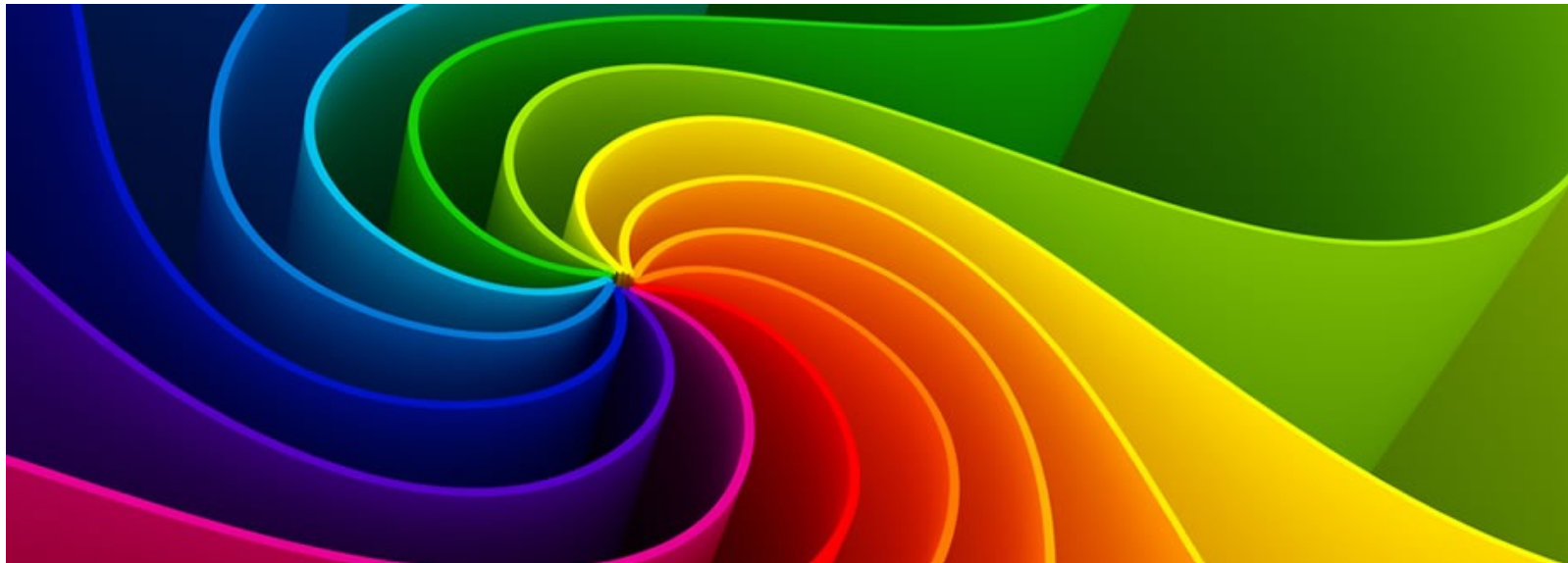


Microservice





Einführung in REST





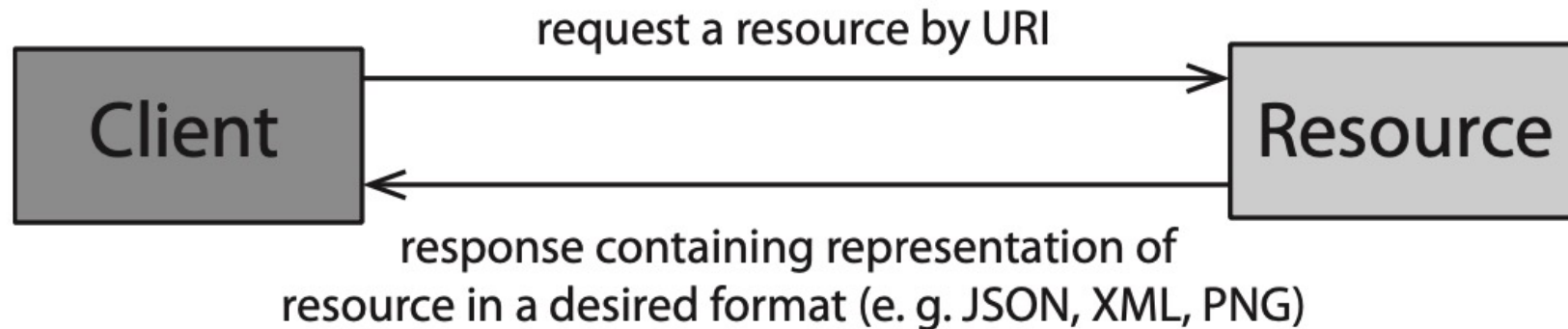
- Heutzutage ist es weit verbreitet, dass Applikationen als verteilte Systeme realisiert werden und verschiedene Systembestandteile auf unterschiedlichen Rechnern oder virtuellen Maschinen (VMs) laufen.
 - Zur Bereitstellung von Funktionalität über das Netzwerk kann man sogenannte Webservices einsetzen, also Dienste, die auf das Web ausgerichtet sind.
 - Im Gegensatz zu den klassischen, in einem Browser laufenden Webapplikationen sind Webservices weniger für menschliche Benutzer gedacht, sondern bieten Funktionalitäten in Form spezieller APIs (wie Twitter, Google usw.) an, die von anderen Programmen zugreifbar sind.
 - REST = REpresentational State Transfer
-



- REST = REpresentational State Transfer
 - Die grundsätzliche **Idee** besteht darin, alle Funktionalität in Form von **adressierbaren Ressourcen bereitzustellen** – und eben nicht über Methoden. Eine Ressource kann alles sein. Auf sie kann über einen Uniform Resource Identifier (URI) zugegriffen werden.
 - **REST nutzt HTTP für eine zustandslose Client-Server-Kommunikation.** Dabei sendet ein Client seine Anfragen (*Requests*) an einen Server, der diese bearbeitet und dann Antworten (*Responses*) zurücksendet.
 - **Es wird kein Standardformat für den Nachrichtenaustausch definiert.**
 - Wir können REST sowohl mit XML als auch mit JSON erstellen. JSON ist gebräuchlicher
-



- Man unterscheidet zwischen einem **REST-Server**, der Ressourcen bereitstellt, und **REST-Clients**, die auf diese Ressourcen zugreifen. Dazu besitzt jede Ressource eine ID, die in der Regel als URI (Uniform Resource Identifier) modelliert ist.



- Zur Adressierung eines *REST-Service* (auch *Ressource* genannt) dient eine URI, die aus Server und Port sowie einem Basispfad und einem Pfad der Ressource besteht:

```
http://server:port/basePath/resourcePath/
```



- Zur Adressierung eines *REST-Service* (auch *Ressource* genannt) dient eine **URI**, die aus **Server und Port** sowie einem **Basispfad** und einem **Pfad der Ressource** besteht:

```
http://server:port/basePath/resourcePath/
```

Damit wird der darunter registrierte REST-Service angesprochen, der die gewünschte Aktion ausführt. Es werden z. B. neue Datensätze angelegt oder Informationen zu bestehenden Datensätzen abgefragt. Die Kommunikation basiert auf HTTP und stützt sich vor allem auf die vier Operationen **POST**, **GET**, **PUT** und **DELETE**:

- **POST** – Erzeugt neue Daten, d. h. eine neue Ressource.
 - **GET** – Definiert einen Lesezugriff auf eine oder mehrere Ressourcen.
 - **PUT** – Verändert eine existierende Ressource. Falls diese noch nicht existiert, kann eine Ressource auch neu erzeugt werden.
 - **DELETE** – Löscht eine Ressource.
-



HTTP-Befehl	URL-Pfad	Beschreibung
POST	<code>/customers</code>	Erzeugt einen neuen Kunden mithilfe der Informationen des Bodys.
GET	<code>/customers</code>	Ermittelt alle Kunden.
GET	<code>/customers/<id></code>	Ermittelt den Kunden mit der im Pfad übergebenen <code>id</code> .
PUT	<code>/customers/<id></code>	Aktualisiert den Datensatz des Kunden mit der übergebenen <code>id</code> mithilfe der Informationen des Bodys.
DELETE	<code>/customers/<id></code>	Löscht den Kunden mit der Id <code>id</code> .

Typische HTTP-Status-Codes



200	Ok	Die Anfrage wurde erfolgreich bearbeitet
201	Created	Eine Ressource wurde erfolgreich an den Server geschickt und erstellt
204	No Content	Die Anfrage wurde erfolgreich bearbeitet, jedoch enthält die Antwort keine Daten. Wird z. B. oft bei DELETE verwendet
400	Bad Request	Die Anfrage wurde nicht richtig aufgebaut und dem Server fehlen Daten
401	Unauthorized	Die Anfrage benötigt eine Authentifizierung
403	Forbidden	Der Client hat nicht die nötigen Rechte, um eine Operation durchzuführen
404	Not Found	Die angeforderte Ressource wurde nicht gefunden
500	Internal Server Error	Serverseitig kommt es zu einem Fehler und die Anfrage kann nicht richtig bearbeitet werden



RestService
to say
“hello”

@SpringBootApplication

```
public class App {
```

```
    public static void main(String[] args) {  
        SpringApplication.run(App.class, args);
```

```
    }
```

```
}
```

@RestController

```
public class MyRestController{
```

```
    @GetMapping("/hello")
```

```
    public String hello(){  
        return "hello hbt";
```

```
    }
```

```
}
```

RECAP: More REST actions



```
@GetMapping("/items")
public void.getItems(){
    ...
}
```

```
@GetMapping("/items/{itemId}")
public void.getItems(@PathVariable String itemId){
    ...
}
```

```
@PostMapping("/items")
public void.putItems(@RequestBody ShoppingItem item){
    ...
}
```

Beispiel: Typischer CRUD-REST-Server



```
@RestController
public class ProductController {
    @Autowired
    private IProductService productService;

    @PostMapping(value = "/products")
    public Product create(@RequestBody Product product) {
        return productService.create(product);
    }

    @GetMapping(value = "/products")
    public List<Product> getProduct() {
        return productService.findAll();
    }
}
```

...

Beispiel: Typischer CRUD-REST-Server



...

```
@GetMapping(value = "/products/{id}")  
public Product findById(@PathVariable("id") long id) {  
    return productService.findById(id);  
}
```

```
@DeleteMapping(value = "/products/{id}")  
public void deleteById(@PathVariable("id") long id) {  
    productService.deleteById(id);  
}
```

```
}
```



DEMO

«spring-rest-product-controller»

Beispiel: Spring Boot Rest App



- Verwaltung von Büchern
- RESTController mit CRUD-Funktionalität
- In-Memory-Fake-Repository
- Aber mit Validation!

New Spring Starter Project Dependencies

Spring Boot Version: 2.5.6

Available:

- REST
- Developer Tools
 - ☒ Spring Boot DevTools
- NoSQL
 - ☐ Spring Data Elasticsearch (Access+Driver)
- Spring Cloud Discovery
 - ☐ Eureka Discovery Client
- Spring Cloud Routing
 - ☐ OpenFeign
- Testing
 - ☐ Spring REST Docs
 - ☐ Contract Stub Runner

Selected:

- X Spring Boot DevTools
- X Spring Web

< Back Next > Cancel Finish



```
@RestController
@RequestMapping("/books")
public class BookController {

    @Autowired
    private BookService service;

    @GetMapping
    public List<Book> findAll() {
        return service.findAll();
    }

    @GetMapping(value =("/{id}")
    public Book findById(@PathVariable("id") long id) {
        return service.findById(id);
    }

    ...
}
```



```
@RestController
@RequestMapping("/books")
public class BookController {

...

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Book create(@Valid @RequestBody Book resource) {
        Objects.requireNonNull(resource);

        return service.create(resource);
    }

...
}
```



DEMO

«BookManagementApp»



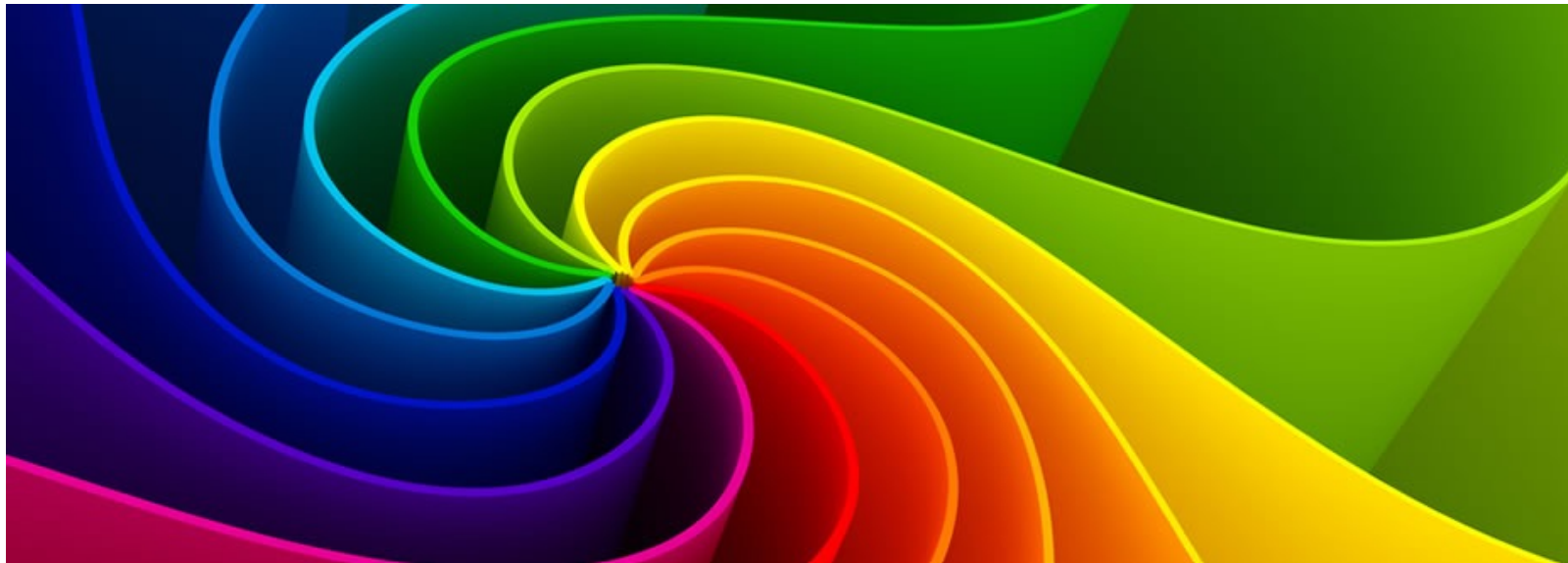
- **RESTful-Webdienste sind plattformunabhängig.**
 - **Sie können in jeder Programmiersprache geschrieben und auf jeder Plattform ausgeführt werden.**
 - **Sie bieten verschiedene Datenformate wie JSON, Text, HTML und XML.**
 - **Die Schnittstelle ist einheitlich und stellt Ressourcen zur Verfügung.**
 - **Der Dienst sollte eine mehrschichtige Architektur aufweisen.**
-

Vergleich mit JAX-RS

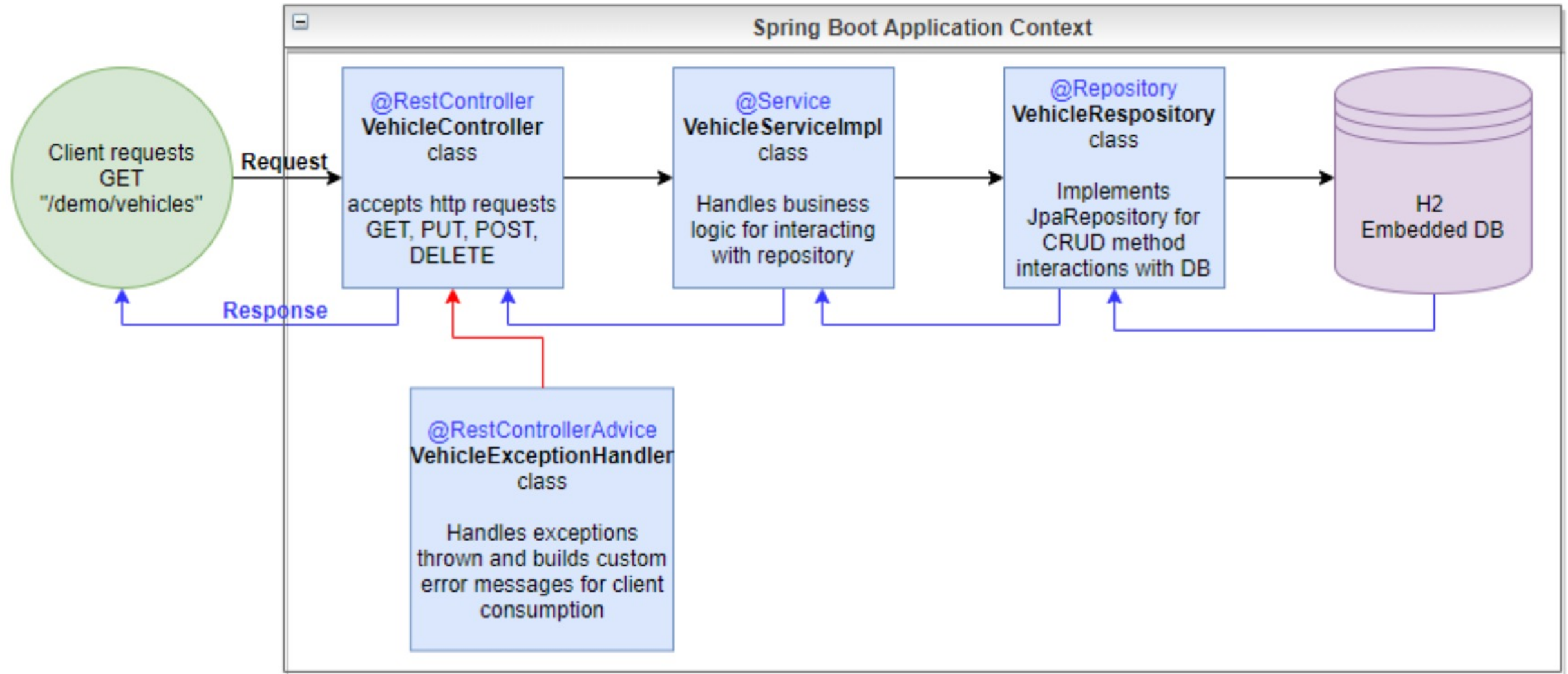
Annotation	Beschreibung
@Path	Legt den Pfad fest, unter dem die Ressource ansprechbar ist.
@POST	Die annotierte Methode reagiert auf ein HTTP <code>POST</code> .
@GET	Die annotierte Methode bearbeitet einen HTTP <code>GET</code> Request.
@PUT	Die annotierte Methode reagiert auf HTTP <code>PUT</code> .
@DELETE	Die annotierte Methode behandelt HTTP <code>DELETE</code> .
@Produces	Bestimmt, welche Rückgabeformate von der annotierten Methode produziert werden können.
@Consumes	Legt für Eingabeparameter fest, in welchem Format diese von der annotierten Methode entgegengenommen werden können.
@PathParam	Beschreibt Parameter, die im Pfad der URL notiert werden.
@QueryParam	Beschreibt Parameter, die im Query-Teil der URL angegeben sind, also nach dem <code>?</code> als Name-Wert-Paar <code>name = value</code> und durch <code>&</code> voneinander getrennt.
@FormParam	Beschreibt Parameter, die über HTML-Formulare eingegeben werden.



Einführung in REST Client



Big picture



Spring Boot REST Client App



- Abfrage von Büchern der anderen App
- RestTemplate zum «Absetzen» von REST-Calls
 - getObject
 - postForObject
 - ...

```
public PhonebookEntry createEntry(PhonebookEntry entry)
{
    final String uri = "http://localhost:8081/api/v1/phonebook";

    RestTemplate restTemplate = new RestTemplate();
    PhonebookEntry result = restTemplate.postForObject( uri, entry, PhonebookEntry.class);
    return result;
}
```

Spring Boot REST Client App



@Component

```
public class BookInfoClient {
```

```
    public static final String SERVER_URI = "http://localhost:8888/books";
```

```
    // ACHTUNG: geht nicht: @Autowired
```

```
    private RestTemplate restTemplate;
```

@Autowired

```
    public BookInfoClient(RestTemplateBuilder builder) {
```

```
        this.restTemplate = builder.build();
```

```
    }
```

@SuppressWarnings("unchecked")

```
    public List<Book> getAllBooks() {
```

```
        List<Book> books = (List<Book>) restTemplate.getForObject(SERVER_URI,  
                                                                    List.class);
```

```
        System.out.println(books);
```

```
        return books;
```

```
    }
```




DEMO

«BookInfoClientApp»



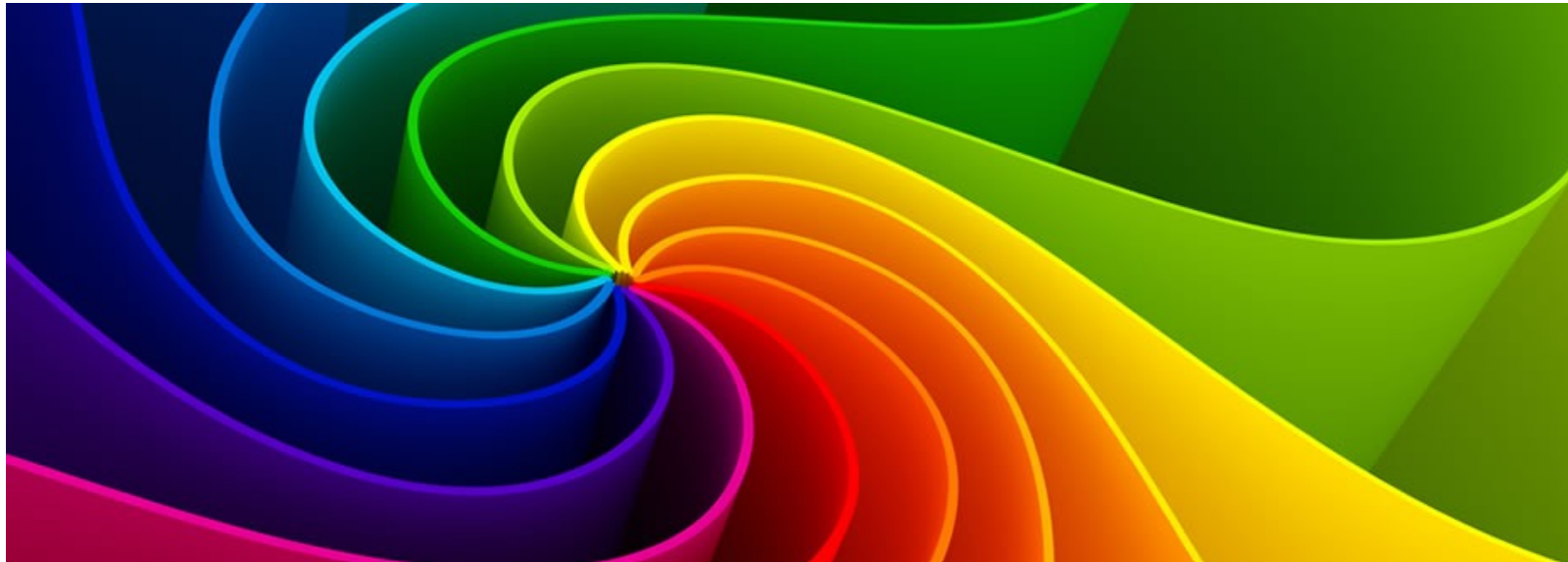
Exercises 1 – 2

https://github.com/Michaeli71/AMTC_Spring_Workshop

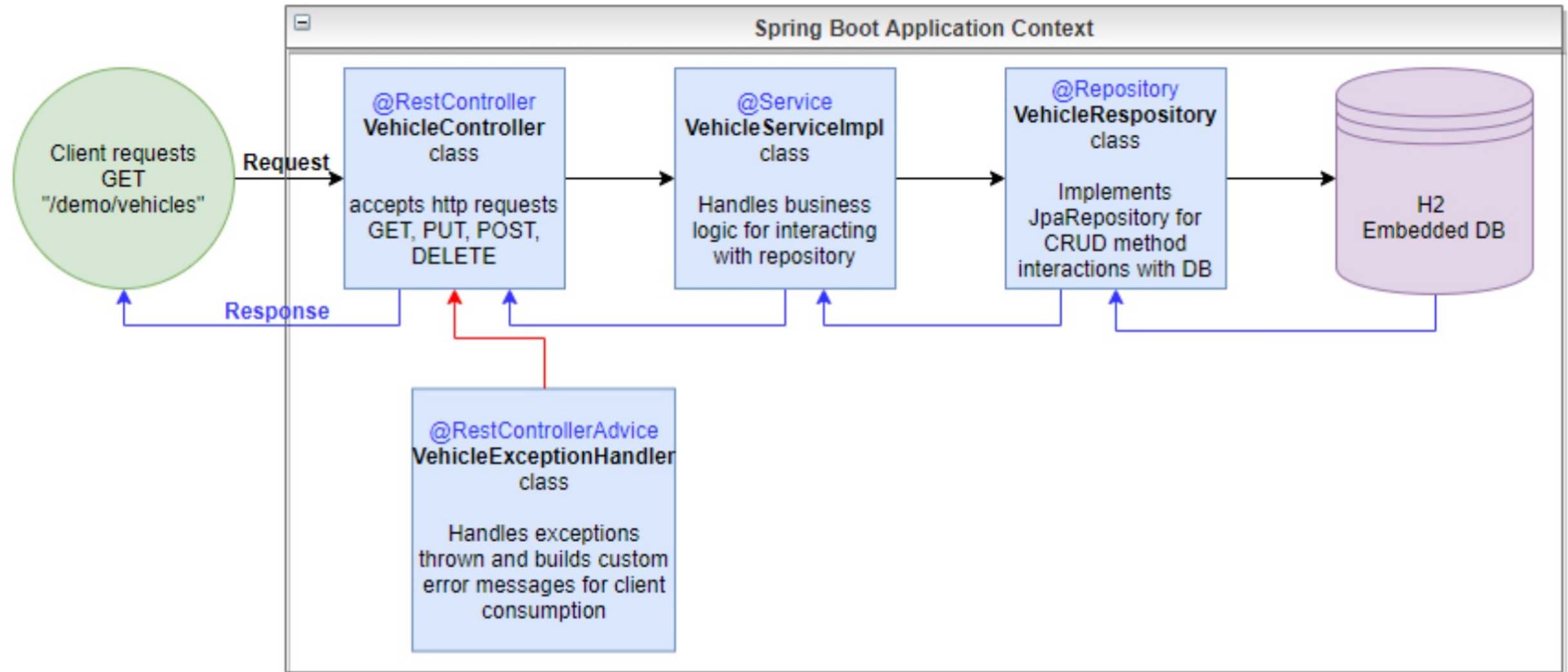




Exception Handler



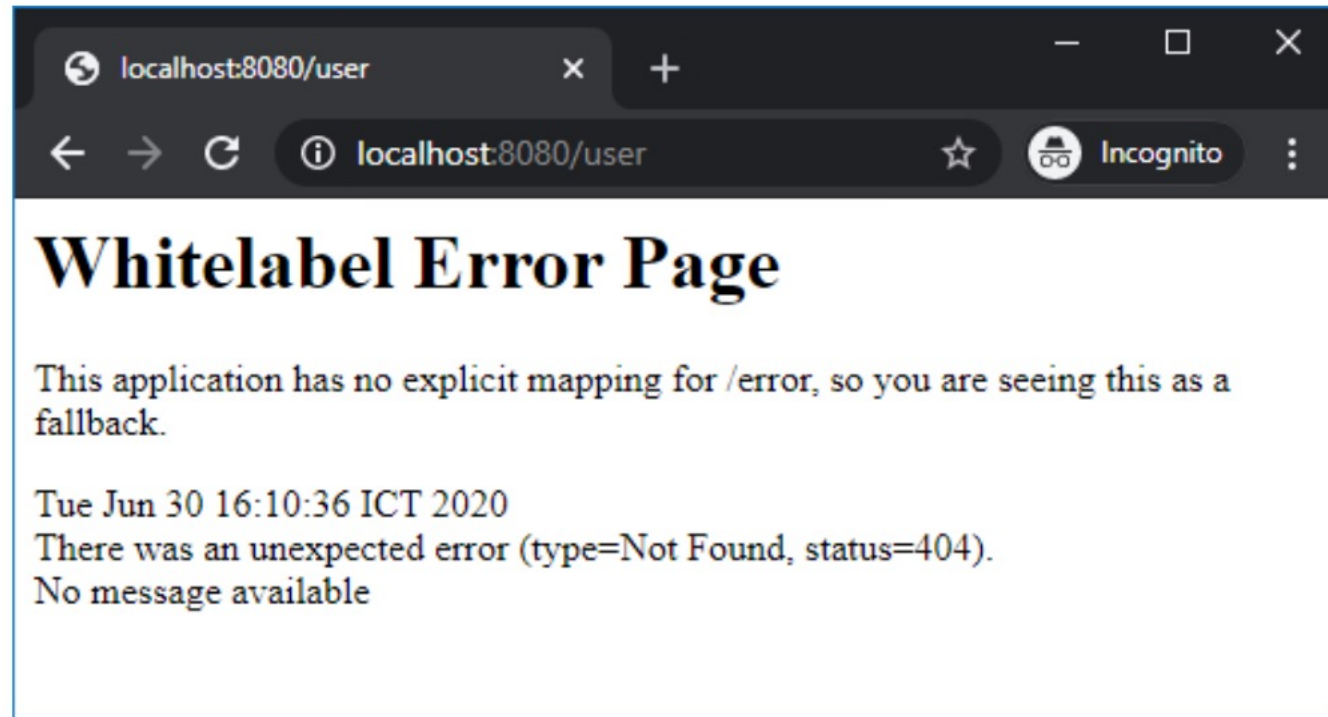
RECAP: More REST actions



Error Handling



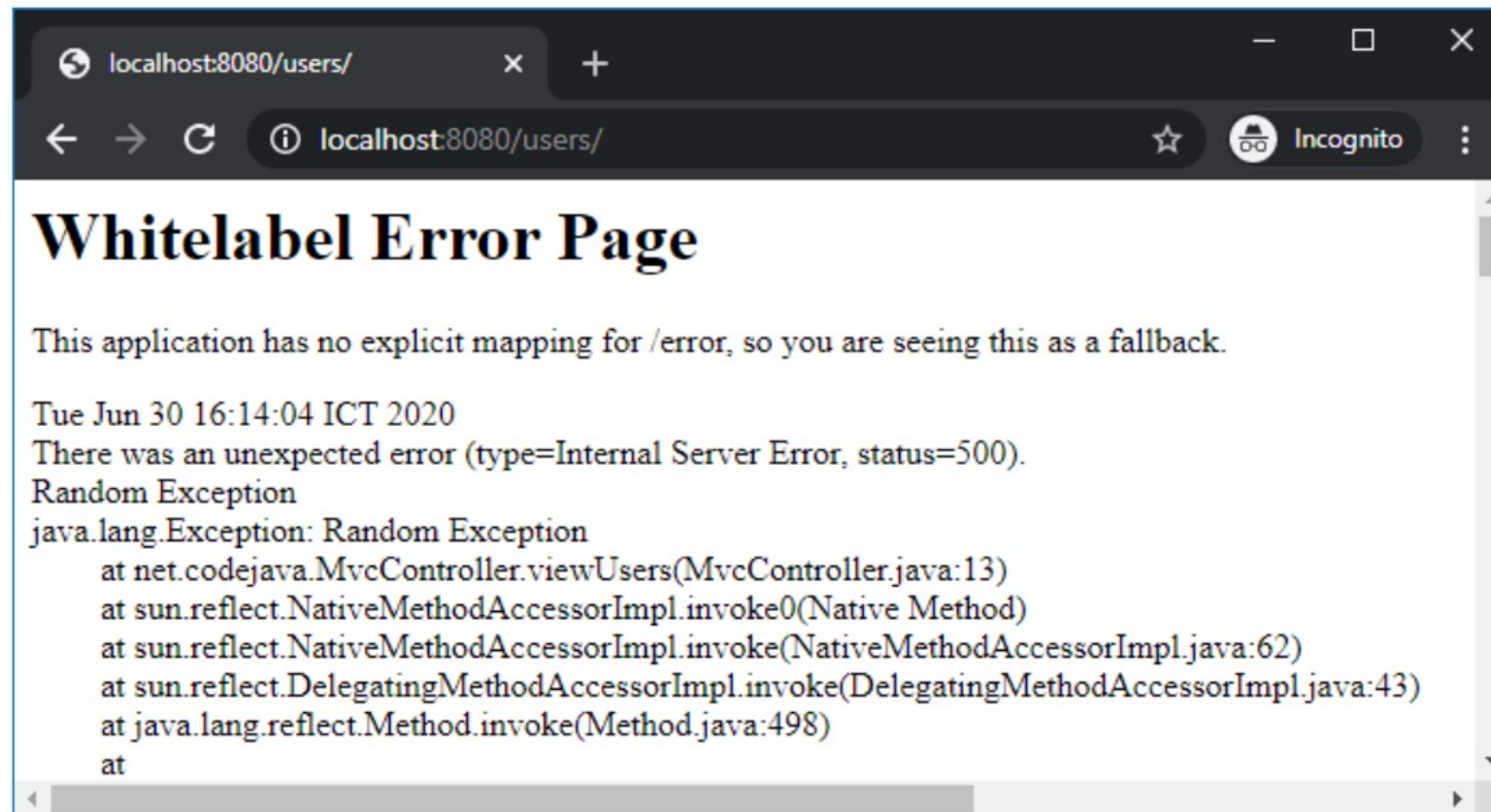
- Standardmäßig zeigt Spring Boot die Whitelabel-Fehlerseite an, wenn ein Fehler auftritt. Zum Beispiel, wenn eine Seite nicht gefunden werden konnte (HTTP 404 Fehler):



Error Handling



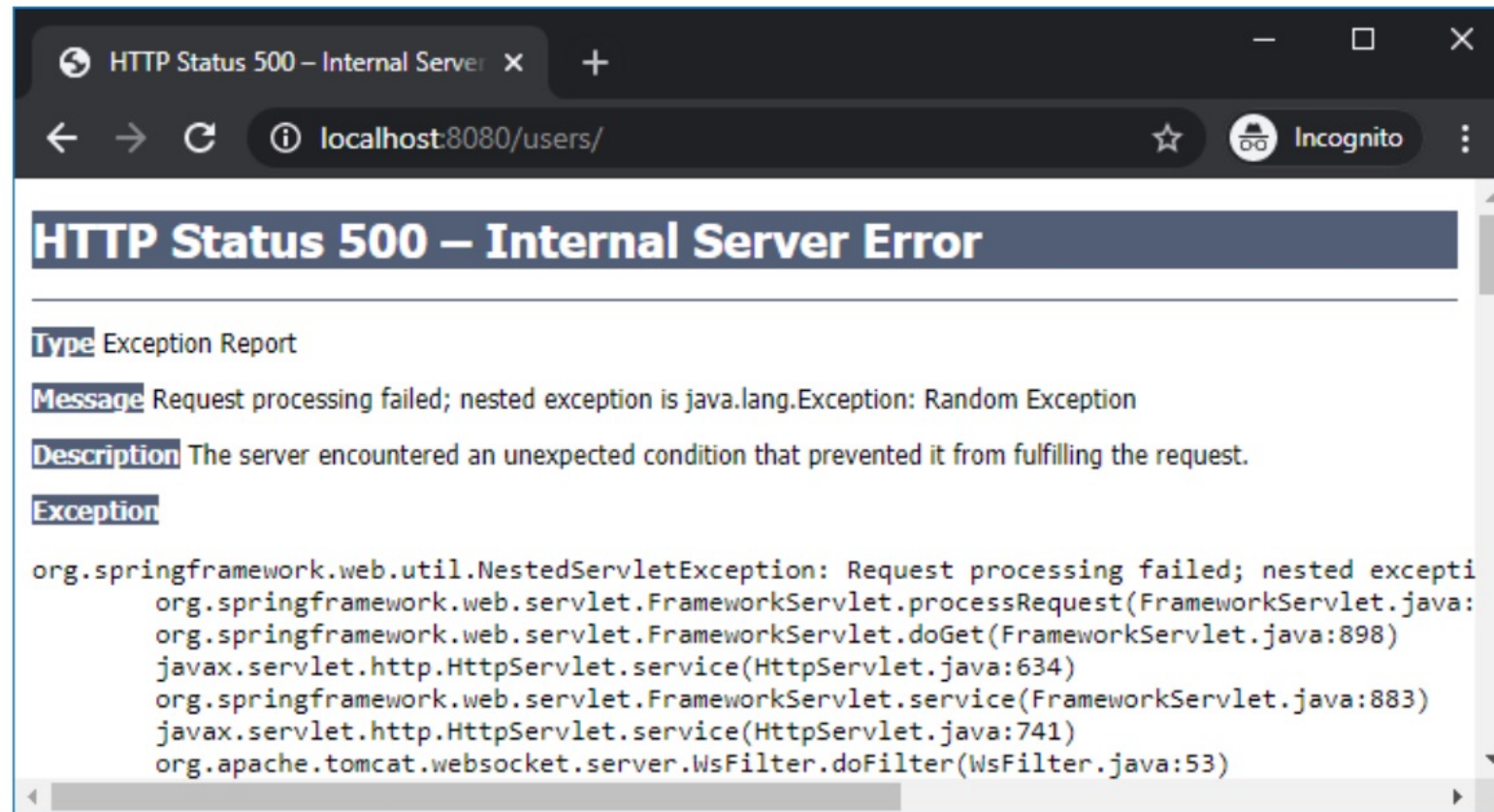
- Wenn eine Exception auftritt, die einen HTTP 500 Internal Server Error verursacht, wird die White-Label-Fehlerseite mit dem Stack-Trace der Ausnahme angezeigt:



Error Handling



- `server.error.whitelabel.enabled=false`

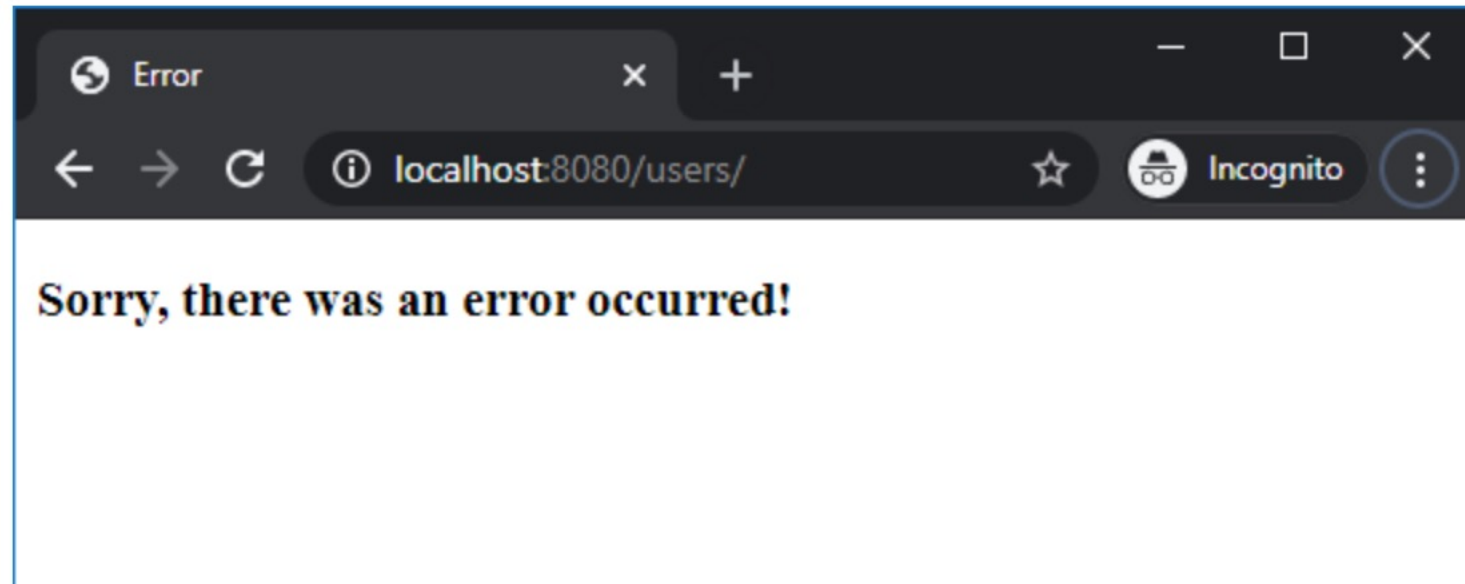


Eigene Error-Seiten



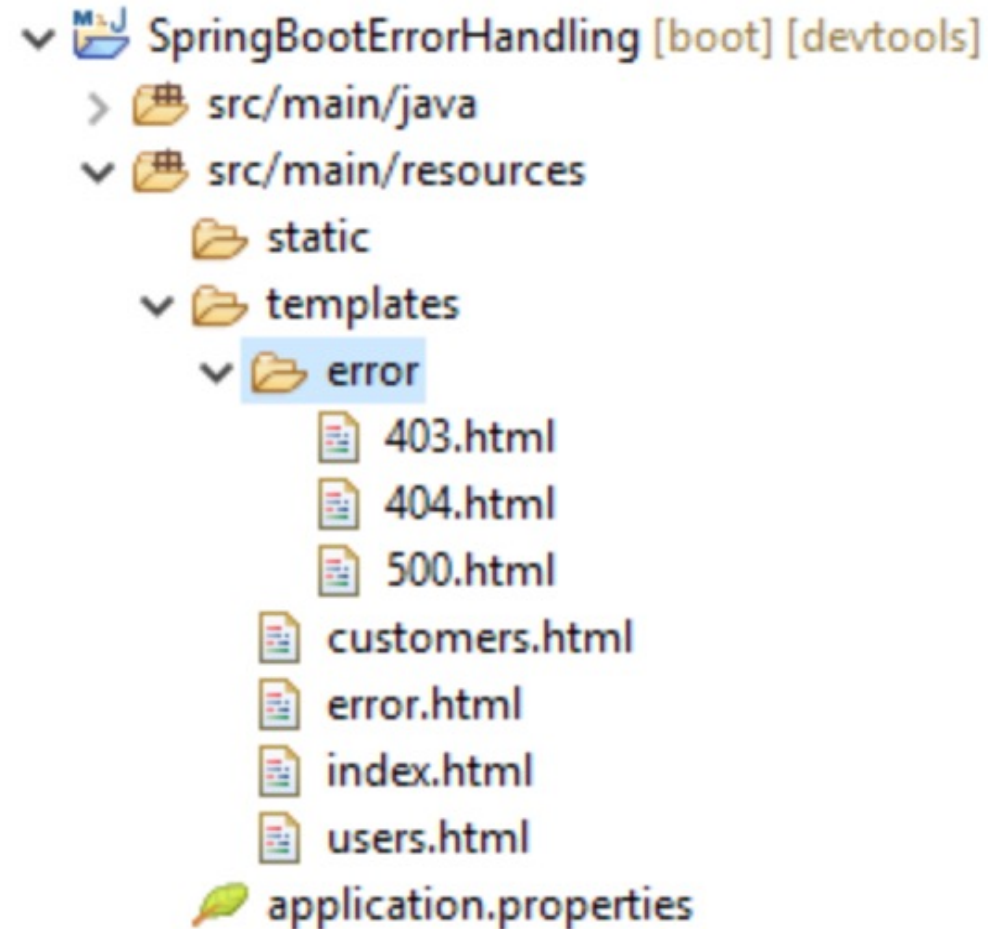
- `server.error.whitelabel.enabled=false`

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Error</title>
</head>
<body>
  <h3>Sorry, there was an error occurred!</h3>
</body>
</html>
```





- Spezifische Error-Seiten





```
<!DOCTYPE html>
<html>
<body>
<h1>Something went wrong! </h1>
<h2>Our Engineers are on it</h2>
<a href="/">Go Home</a>
</body>
</html>
```



localhost:8080/product/123

Something went wrong!

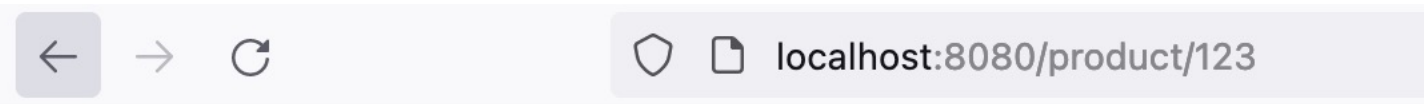
Our Engineers are on it

[Go Home](#)

Eigener Error-Controller



```
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>400 Bad Request</title>
  <meta name="viewport" content="width=device-width">
  <style>
    #error {
      border-color: darkred;
      background-color: aliceblue;
    }
    h2 {
      color: green;
    }
  </style>
</head>
<body>
<div class="error-page-wrap">
  <div id="error">
    <h2>400! oops! Passed WRONG parameters</h2>
  </div>
</div>
</body>
</html>
```



404! oops! Requested page not found



DEMO

«sprint-rest-product-error-controller»

Eigener Error-Controller analog zum Standard



- Wenn Sie einige Aktionen ausführen möchten, bevor die benutzerdefinierten Fehlerseiten angezeigt werden, können Sie eine Controllerklasse implementieren, die das **ErrorHandler-Interface** erfüllt:

```
@RestController
public class CustomErrorHandler implements ErrorHandler {
    @RequestMapping("/error")
    public ModelAndView handleError(HttpServletRequest response) {
        ModelAndView modelAndView = new ModelAndView();

        if (response.getStatus() == HttpStatus.BAD_REQUEST.value()) {
            modelAndView.setViewName("error/400");
        } else if (response.getStatus() == HttpStatus.NOT_FOUND.value()) {
            modelAndView.setViewName("error/404");
        } else if (response.getStatus() == HttpStatus.INTERNAL_SERVER_ERROR.value()) {
            modelAndView.setViewName("error/500");
        } else {
            modelAndView.setViewName("error");
        }
        return modelAndView;
    }
}
```

Ausgangslage: REST Controller mit Exception Handling in Methoden



```
public class DogsController {
    @Autowired private final DogsService service;

    @GetMapping
    public ResponseEntity<List<Dog>> getDogs() {
        List<Dog> dogs;

        try {
            dogs = service.getDogs();
        } catch (DogsServiceException ex) {
            return new ResponseEntity<>(null, null, HttpStatus.INTERNAL_SERVER_ERROR);
        } catch (DogsNotFoundException ex) {
            return new ResponseEntity<>(null, null, HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(dogs, HttpStatus.OK);
    }
}
```



2. Solution 1: the Controller-Level *@ExceptionHandler*

The first solution works at the *@Controller* level. We will define a method to handle exceptions and annotate that with *@ExceptionHandler*.

```
public class FooController{  
  
    //...  
    @ExceptionHandler({ CustomException1.class, CustomException2.class })  
    public void handleException() {  
        //  
    }  
}
```

This approach has a major drawback: **The *@ExceptionHandler* annotated method is only active for that particular Controller**, not globally for the entire application. Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.

4. Solution 3: *@ControllerAdvice*

Spring 3.2 brings support for a **global *@ExceptionHandler* with the *@ControllerAdvice* annotation.**

This enables a mechanism that breaks away from the older MVC model and makes use of *ResponseEntity* along with the type safety and flexibility of *@ExceptionHandler*.

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
    extends ResponseEntityExceptionHandler {

    @ExceptionHandler(value
        = { IllegalArgumentException.class, IllegalStateException.class })
    protected ResponseEntity<Object> handleConflict(
        RuntimeException ex, WebRequest request) {
        String bodyOfResponse = "This should be application specific";
        return handleExceptionInternal(ex, bodyOfResponse,
            new HttpHeaders(), HttpStatus.CONFLICT, request);
    }
}
```

The *@ControllerAdvice* annotation allows us to **consolidate our multiple, scattered *@ExceptionHandler*s from before into a single, global error handling component.**

Allgemeines Exception Handling

@ControllerAdvice

@Slf4j

```
public class DogsServiceErrorAdvice {
```

```
    @ExceptionHandler({RuntimeException.class})
```

```
    public ResponseEntity<String> handleRunTimeException(RuntimeException e) {
```

```
        return error(INTERNAL_SERVER_ERROR, e);
```

```
    }
```

```
    @ExceptionHandler({DogsNotFoundException.class})
```

```
    public ResponseEntity<String> handleNotFoundException(DogsNotFoundException e) {
```

```
        return error(NOT_FOUND, e);
```

```
    }
```

```
    @ExceptionHandler({DogsServiceException.class})
```

```
    public ResponseEntity<String> handleDogsServiceException(DogsServiceException e){
```

```
        return error(INTERNAL_SERVER_ERROR, e);
```

```
    }
```

```
    private ResponseEntity<String> error(HttpStatus status, Exception e) {
```

```
        log.error("Exception : ", e);
```

```
        return ResponseEntity.status(status).body(e.getMessage());
```

```
    }
```

```
}
```



Eigener Error-Controller



```
@ControllerAdvice
public class ProductNotFoundAdvice {
    @ExceptionHandler
    void handleIllegalArgumentException(IllegalArgumentException e,
                                       HttpServletResponse response)
        throws IOException {
        response.sendError(HttpStatus.CONFLICT.value());
    }

    @ExceptionHandler
    void handleIllegalArgumentException(ProductNotFoundException e,
                                       HttpServletResponse response)
        throws IOException {
        response.sendError(HttpStatus.BAD_REQUEST.value());
    }
}
```



DEMO

«sprint-rest-product-error-controller»



Exercises 3 – 4

https://github.com/Michaeli71/AMTC_Spring_Workshop





Questions?



REST

- <https://entwickler.de/spring/spring-boot-tutorial-so-entwickelt-man-rest-services-mit-spring-boot/>
 - <https://spring.io/guides/tutorials/rest/>
 - <https://spring.io/guides/gs/rest-service/>
 - <https://www.techiedelight.com/display-custom-error-pages-in-spring-boot/>
 - <https://www.javadevjournal.com/rest-with-spring-series/>
 - <https://www.javadevjournal.com/spring/exception-handling-for-rest-with-spring/>
 - <https://auth0.com/blog/spring-data-rest-tutorial-developing-rest-apis-with-ease/>
-



EXCEPTION HANDLING

- <https://reflectoring.io/spring-boot-exception-handling/>
 - <https://www.bezkoder.com/spring-boot-restcontrolleradvice/>
 - <https://dzone.com/articles/spring-rest-service-exception-handling-1>
 - <https://www.codejava.net/frameworks/spring-boot/spring-boot-error-handling-guide>
-



Thank You
