RESTful Web Services in Spring Boot

In diesem Modul

- REST-Controller Basics: ResponseEntity, Content Negotiation
- Validation & Error Handling
- Moderne HTTP Clients: RestClient, Declarative HTTP Interfaces
- Async/Streaming: CompletableFuture, Server-Sent Events
- Response Wrapping mit @ControllerAdvice
- File Upload (Multipart)
- API Versioning Strategien
- Virtual Threads (Project Loom)

Wiederholung: Der @RestController

- Spezielle `Controller`-Annotation, die `Controller` und `@ResponseBody` kombiniert.
- Jede Methode gibt direkt Daten zurück (keine View-Auflösung).
- Behandelt JSON/XML-Serialisierung automatisch.

```java
@RestController
@RequestMapping("/api/v1/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        // ... Logik
        return new User(id, "Alice");
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED) // Setzt den HTTP Status 201
    public User createUser(@RequestBody User user) {
        // ... Logik
        return user;
    }
}
```

Wiederholung: ResponseEntity

Volle Kontrolle über HTTP Response (Status, Header, Body).

```
@GetMapping("/{id}")
public ResponseEntity<User> findUser(@PathVariable Long id) {
    Optional<User> user = userService.findById(id);
    return user.map(ResponseEntity::ok) // 200 OK
               .orElse(ResponseEntity.notFound().build()); // 404 Not Found
}
```

DTO-Validierung

Das Bean Validation-Framework wird auch vom Spring Framework unterstützt, um Daten mit verschiedenen Annotationen zu prüfen.
Die wichtigsten Annotationen:

- `@Valid`: Standard-JSR 380 (Bean Validation) Annotation.
- `@Validated`: Spring-spezifisch, unterstützt Validation Groups.

DTO-Validierung

```java
// DTO für die Anfrage
public class UserCreateDto {
    @NotBlank(message = "Name darf nicht leer sein")
    @Size(min = 3, max = 50, message = "Name muss 3-50 Zeichen haben")
    private String name;

    @Email(message = "Ungültiges E-Mail Format")
    private String email;

    // Getter & Setter
}

@PostMapping
public ResponseEntity<User> createUser(@Valid @RequestBody UserCreateDto userDto) {
    // Wenn Validation fehlschlägt, wird eine MethodArgumentNotValidException geworfen
    // ...
}
```

Validation Groups

Unterschiedliche Regeln für verschiedene Szenarien (z.B. Erstellen vs. Aktualisieren).

```
// Interfaces als Marker
public interface OnCreate {}
public interface OnUpdate {}

public class UserDto {
    @NotNull(groups = OnUpdate.class) // Nur bei Update nötig
    private Long id;

    @NotBlank(groups = OnCreate.class) // Nur bei Create nötig
    private String name;
}

@PostMapping
public ResponseEntity<User> create(@Validated(OnCreate.class) @RequestBody UserDto dto) { ... }

@PutMapping
public ResponseEntity<User> update(@Validated(OnUpdate.class) @RequestBody UserDto dto) { ... }
```

Custom Validators

Eigene Validierungslogik implementieren.

```
@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = UniqueEmailValidator.class)
public @interface UniqueEmail {
    String message() default "E-Mail bereits vergeben";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class UniqueEmailValidator implements ConstraintValidator<UniqueEmail, String> {
    @Autowired private UserRepository userRepository;

    @Override
    public boolean isValid(String email, ConstraintValidatorContext context) {
        return userRepository.findByEmail(email).isEmpty();
    }
}
```

Globales Error Handling

Zentrales Fehlerhandling für die gesamte REST-API.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ProblemDetail handleValidationExceptions(MethodArgumentNotValidException ex) {
        ProblemDetail pd = ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST, "Validierungsfehler");
        pd.setTitle("Invalid Request Body");
        return pd;
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ProblemDetail handleNotFound(ResourceNotFoundException ex) {
        return ProblemDetail.forStatusAndDetail(
            HttpStatus.NOT_FOUND, ex.getMessage());
    }
}
```

ProblemDetails (RFC 7807)

Standardisiertes Format für HTTP API Fehlerantworten (seit Spring Boot 3).

```
{
  "type": "about:blank",
  "title": "Invalid Request Body",
  "status": 400,
  "detail": "Validierungsfehler",
  "instance": "/api/v1/users",
  "errors": {
    "name": "Name darf nicht leer sein",
    "email": "Ungültiges E-Mail Format"
  }
}
```

Spring Boot konvertiert `ProblemDetail` automatisch in JSON oder XML, wenn der `Accept`-Header dies verlangt.

ProblemDetails erweitern

Oft reicht der Standard nicht. Wir wollen z.B. eine `traceId` oder spezifische Business-Error-Codes hinzufügen.

```
@ExceptionHandler(MyBusinessException.class)
public ProblemDetail handleBusinessException(MyBusinessException ex) {
    ProblemDetail pd = ProblemDetail.forStatusAndDetail(
        HttpStatus.BAD_REQUEST, ex.getMessage());

    // Eigene Properties hinzufügen
    pd.setProperty("errorCode", "BUS-001");
    // Angenommen wir haben einen Tracer injectet
    pd.setProperty("traceId", tracer.currentSpan().context().traceId());

    return pd;
}
```

Moderne HTTP Clients

Status Quo: RestTemplate

- Lange Zeit der Standard für synchrone Calls.
- Jetzt im **Maintenance Mode**. Es wird keine neuen Features mehr geben.
- Nachteil: Viele überladene Methoden, kein Fluent API.

Die Nachfolger:

1. **RestClient (Spring Boot 3.2):** Synchron, Fluent API. Basiert auf Servlet-Stack.
2. **WebClient (Spring 5):** Reaktiv, non-blocking. Erfordert `spring-boot-starter-webflux`.
3. **Declarative HTTP Interfaces (Spring 6):** Interface-basiert (via Proxy).

Der RestClient (Synchron)

Bietet eine moderne Fluent API ohne Reactive Stack (Mono/Flux).

```
@Service
public class ProductClient {
    private final RestClient restClient;

    public ProductClient(RestClient.Builder builder) {
        this.restClient = builder.baseUrl("https://api.example.com").build();
    }

    public Product getProduct(String id) {
        return restClient.get()
            .uri("/products/{id}", id)
            .retrieve()
            .body(Product.class);
    }
}
```

Declarative HTTP Interfaces

Definiere die API als Java Interface (ähnlich Feign/Retrofit).

```
public interface UserApi {
    @GetExchange("/users/{id}")
    User getById(@PathVariable Long id);

    @PostExchange("/users")
    void createUser(@RequestBody User user);
}
```

Dies benötigt einen **Unterbau**, der die Requests ausführt (WebClient oder RestClient).

Declarative Client Factory (mit RestClient)

Verbindung von Interface und Engine.

```
@Configuration
public class ClientConfig {
    @Bean
    UserApi userApi(RestClient.Builder builder) {
        RestClient client = builder.baseUrl("https://user-service").build();

        // Nutzt den synchronen RestClient als Engine
        RestClientAdapter adapter = RestClientAdapter.create(client);

        HttpServiceProxyFactory factory = HttpServiceProxyFactory
            .builderFor(adapter)
            .build();

        return factory.createClient(UserApi.class);
    }
}
```

Asynchrone APIs & Streaming

`CompletableFuture` als Rückgabetyp

- Der Controller Thread wird freigegeben, während die Logik im Hintergrund arbeitet.
- Verbessert die Skalierbarkeit bei blockierenden Operationen.

```
@RestController
public class AsyncController {
    @Autowired private SlowService slowService;

    @GetMapping("/async-result")
    public CompletableFuture<String> getAsyncResult() {
        return CompletableFuture.supplyAsync(() -> slowService.doSlowWork());
    }
}
```

Streaming Responses (Server-Sent Events)

Für Realtime-Updates, z.B. wenn der Client ständig neue Daten erhalten soll.

```
@RestController
public class SseController {

    @GetMapping(value = "/events", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public SseEmitter streamEvents() {
        SseEmitter emitter = new SseEmitter();

        new Thread(() -> {
            // Hier: Events asynchron an den Emitter senden
            // z.B. aus einem Message Queue Listener oder einem Scheduled Task
            emitter.send(SseEmitter.event().name("message").data("Hello, Client!"));
            emitter.complete(); // Verbindung schließen
        }).start();

        return emitter;
    }
}
```

OpenAPI (früher Swagger)

Ziel

Standardisierte, maschinenlesbare Beschreibung von REST-APIs.

- **Dokumentation:** Interaktive UI (Swagger UI).
- **Code-Generierung:** Clients in jeder Sprache.

Man unterscheidet zwei Ansätze: Code First und Contract First.

Ansatz 1: Code-First (SpringDoc OpenAPI)

Man schreibt den Code, die Doku wird daraus generiert.

```
@RestController
@RequestMapping("/products")
@Tag(name = "Produktverwaltung", description = "API für CRUD-Operationen an Produkten")
public class ProductController {

    @Operation(
        summary = "Produkt anhand ID abrufen",
        description = "Gibt ein Produktobjekt basierend auf der bereitgestellten ID zurück.",
        parameters = @Parameter(name = "id", description = "ID des Produkts", example = "123")
    )
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Produkt gefunden",
                     content = @Content(schema = @Schema(implementation = Product.class))),
        @ApiResponse(responseCode = "404", description = "Produkt nicht gefunden")
    })
    @GetMapping("/{id}")
    public Product getProduct(@PathVariable Long id) {
        return new Product(id, "Advanced Widget");
    }
}
```

Ansatz 2: Contract-First

Man schreibt zuerst die OpenAPI-Spezifikation (YAML/JSON) und generiert daraus den Code (Interfaces, DTOs).

Vorteile:

- **API-Design als erste Klasse:** Fokus auf das API-Design, bevor implementiert wird.
- **Parallele Entwicklung:** Backend- und Frontend-Teams können gleichzeitig arbeiten.
- **Konsistenz:** API ist über alle Services hinweg konsistent.

Tool: `openapi-generator-maven-plugin` (oder Gradle Plugin).

Response Wrapping mit @ControllerAdvice

ResponseBodyAdvice

Ermöglicht das **globale Wrapping** aller Response Bodies – z.B. für ein einheitliches API-Format.

```
{
  "success": true,
  "data": { ... },      // Der eigentliche Response
  "timestamp": "...",
  "traceId": "..."
}
```

ResponseBodyAdvice Implementierung

```
@ControllerAdvice
public class ApiResponseWrapper implements ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter returnType, Class converterType) {
        // Nur für eigene Controller, nicht für Actuator etc.
        return returnType.getContainingClass().getPackageName()
                         .startsWith("com.example.api");
    }

    @Override
    public Object beforeBodyWrite(Object body, MethodParameter returnType,
                                  MediaType contentType, Class converterType,
                                  ServerHttpRequest request, ServerHttpResponse response) {
        // ProblemDetail nicht wrappen
        if (body instanceof ProblemDetail) return body;

        return new ApiResponse<>(true, body, Instant.now());
    }
}
```

ApiResponse Record

```
public record ApiResponse<T>(
    boolean success,
    T data,
    Instant timestamp
) {
    public ApiResponse(boolean success, T data, Instant timestamp) {
        this.success = success;
        this.data = data;
        this.timestamp = timestamp;
    }
}
```

File Upload

Multipart File Upload

Spring Boot unterstützt Datei-Uploads über `MultipartFile`.

```
@RestController
@RequestMapping("/api/files")
public class FileUploadController {

    @PostMapping("/upload")
    public ResponseEntity<FileInfo> uploadFile(@RequestParam("file") MultipartFile file) {
        if (file.isEmpty()) {
            return ResponseEntity.badRequest().build();
        }

        String filename = StringUtils.cleanPath(file.getOriginalFilename());
        Path targetPath = Paths.get("uploads").resolve(filename);
        Files.copy(file.getInputStream(), targetPath, StandardCopyOption.REPLACE_EXISTING);

        return ResponseEntity.ok(new FileInfo(filename, file.getSize()));
    }
}
```

Konfiguration für große Dateien

```yaml
spring:
  servlet:
    multipart:
      enabled: true
      max-file-size: 10MB        # Max. Größe pro Datei
      max-request-size: 50MB     # Max. Größe des gesamten Requests
      file-size-threshold: 2KB   # Ab dieser Größe auf Disk schreiben
```

Mehrere Dateien hochladen

```
@PostMapping("/upload-multiple")
public ResponseEntity<List<FileInfo>> uploadMultiple(
        @RequestParam("files") List<MultipartFile> files) {

    List<FileInfo> results = files.stream()
        .filter(f -> !f.isEmpty())
        .map(this::saveFile)
        .toList();

    return ResponseEntity.ok(results);
}

@PostMapping("/upload-with-metadata")
public ResponseEntity<FileInfo> uploadWithMetadata(
        @RequestPart("file") MultipartFile file,
        @RequestPart("metadata") FileMetadata metadata) {  // JSON Part

    // file + metadata verarbeiten
    return ResponseEntity.ok(new FileInfo(file.getOriginalFilename(), file.getSize()));
}
```

API Versioning

Warum API Versioning?

- **Breaking Changes:** Alte Clients sollen weiter funktionieren.
- **Parallele Versionen:** v1 und v2 gleichzeitig betreiben.
- **Deprecation:** Sanfte Migration ermöglichen.

Strategie 1: URL Path Versioning

Die Version ist Teil der URL.

```
@RestController
@RequestMapping("/api/v1/users")
public class UserControllerV1 {
    @GetMapping("/{id}")
    public UserV1 getUser(@PathVariable Long id) { ... }
}

@RestController
@RequestMapping("/api/v2/users")
public class UserControllerV2 {
    @GetMapping("/{id}")
    public UserV2 getUser(@PathVariable Long id) { ... }
}
```

Pro: Einfach, klar sichtbar, gut cachebar.

Con: URL-Proliferation, nicht RESTful (Resource ändert sich nicht).

Strategie 2: Header Versioning

Version wird im Header übergeben.

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping(value = "/{id}", headers = "X-API-Version=1")
    public UserV1 getUserV1(@PathVariable Long id) { ... }

    @GetMapping(value = "/{id}", headers = "X-API-Version=2")
    public UserV2 getUserV2(@PathVariable Long id) { ... }
}
```

Pro: Saubere URLs.

Con: Nicht im Browser testbar, Header kann vergessen werden.

Strategie 3: Media Type Versioning

Version im `Accept`-Header (Content Negotiation).

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping(value = "/{id}", produces = "application/vnd.myapi.v1+json")
    public UserV1 getUserV1(@PathVariable Long id) { ... }

    @GetMapping(value = "/{id}", produces = "application/vnd.myapi.v2+json")
    public UserV2 getUserV2(@PathVariable Long id) { ... }
}
```

Pro: RESTful, Resource-URL bleibt stabil.

Con: Komplex, schwer zu testen.

Strategie 4: Query Parameter

```
@GetMapping("/{id}")
public ResponseEntity<?> getUser(
        @PathVariable Long id,
        @RequestParam(defaultValue = "1") int version) {

    return switch (version) {
        case 1 -> ResponseEntity.ok(userService.getUserV1(id));
        case 2 -> ResponseEntity.ok(userService.getUserV2(id));
        default -> ResponseEntity.badRequest().body("Unknown version");
    };
}
```

Pro: Einfach zu testen.

Con: Nicht standardisiert, Query-Params eigentlich für Filter.

Empfehlung

Szenario	Empfohlene Strategie
Öffentliche API	**URL Path** (am klarsten)
Interne Microservices	**Header** oder **Media Type**
Schnelle Iteration	**Query Parameter** (pragmatisch)

Tipp: Egal welche Strategie – dokumentieren Sie Ihre Deprecation-Policy!

Virtual Threads (Project Loom)

Das Problem: Thread-per-Request

Klassische Servlet-Container nutzen einen **Thread pro Request**.

- **Tomcat Default:** ~200 Threads
- **Blockierender Request:** Thread wartet auf DB/API → verschwendet
- **Mehr Throughput?** Mehr Threads → mehr RAM, Context-Switching

Lösung bisher: Reactive Programming (WebFlux) – aber komplexer Code.

Virtual Threads: Die Lösung

Java 21 (LTS) bringt **Virtual Threads** – leichtgewichtige Threads, die vom JVM verwaltet werden.

- **Millionen** von Virtual Threads möglich
- **Blockieren ist OK** – JVM parkt den Virtual Thread
- **Carrier Thread** wird für andere Arbeit freigegeben
- **Kein reaktiver Code nötig** – synchroner Stil funktioniert

Virtual Threads aktivieren

Spring Boot 3.2+:

```
spring:
  threads:
    virtual:
      enabled: true
```

Das war's! Alle Request-Handler laufen jetzt auf Virtual Threads.

Vorher vs. Nachher

Ohne Virtual Threads:

```
Request 1 → Platform Thread 1 (wartet auf DB...)  ← blockiert
Request 2 → Platform Thread 2 (wartet auf DB...)  ← blockiert
Request 3 → Platform Thread 3 ...
...
Request 201 → REJECTED (Thread Pool voll!)
```

Mit Virtual Threads:

```
Request 1 → Virtual Thread 1 (wartet auf DB...)  ← JVM parkt
Request 2 → Virtual Thread 2 (wartet auf DB...)  ← JVM parkt
...
Request 10000 → Virtual Thread 10000  ← Kein Problem!
```

Wann Virtual Threads nutzen?

Szenario	Empfehlung
I/O-lastige Anwendung (DB, HTTP)	Virtual Threads
CPU-lastige Berechnung	Platform Threads
Bestehendes WebFlux	Kein Vorteil (bereits non-blocking)
Legacy-Code mit `synchronized`	Testen! (Pinning-Problem)

Das Pinning-Problem

Virtual Threads können **gepinnt** werden, wenn sie einen `synchronized`-Block betreten.

```
// Problematisch: Virtual Thread wird an Carrier gepinnt
synchronized (lock) {
    blockingDatabaseCall();  // Carrier Thread blockiert!
}

// Besser: ReentrantLock verwenden
lock.lock();
try {
    blockingDatabaseCall();  // Virtual Thread kann yielden
} finally {
    lock.unlock();
}
```

Diagnose: `-Djdk.tracePinnedThreads=short`

Virtual Threads mit @Async

Auch `@Async`-Methoden können Virtual Threads nutzen:

```
@Configuration
@EnableAsync
public class AsyncConfig {

    @Bean
    public Executor taskExecutor() {
        return Executors.newVirtualThreadPerTaskExecutor();
    }
}
```

```
@Service
public class EmailService {

    @Async
    public CompletableFuture<Void> sendEmailAsync(String to, String content) {
        // Läuft auf Virtual Thread
        emailClient.send(to, content);
```

Virtual Threads: Caveats

1. **ThreadLocal:** Vorsicht bei großem ThreadLocal-Speicher (Millionen Threads!)
2. **Native Code:** JNI-Calls können Virtual Threads pinnen
3. **Monitoring:** Thread-Dumps zeigen sehr viele Threads
4. **Connection Pools:** Können zum Bottleneck werden (Pool < Virtual Threads)

```
# Connection Pool anpassen
spring:
  datasource:
    hikari:
      maximum-pool-size: 50  # Erhöhen, aber DB-Limits beachten!
```

Structured Concurrency (Preview)

Java 21+ bietet auch **Structured Concurrency** für parallele Tasks:

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Supplier<User> userTask = scope.fork(() -> userService.getUser(id));
    Supplier<List<Order>> ordersTask = scope.fork(() -> orderService.getOrders(id));

    scope.join();           // Warte auf alle
    scope.throwIfFailed();  // Exception bei Fehler

    return new UserProfile(userTask.get(), ordersTask.get());
}
```

Vorteil: Alle Subtasks werden bei Fehler automatisch abgebrochen.