Modul 10 - REST Adapter

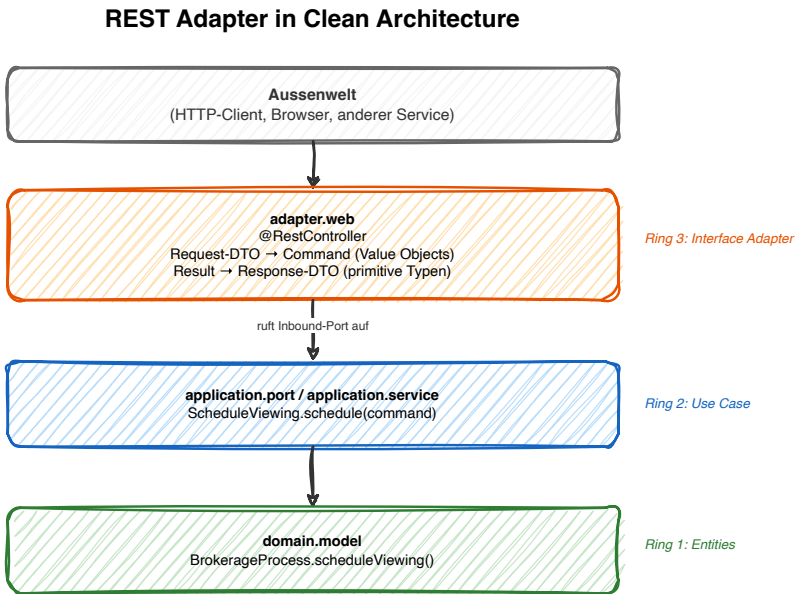
Der Inbound Adapter in Clean Architecture

Geschätzte Dauer: ca. 70 Minuten

Lernziele

- `@RestController` als Inbound Adapter in Clean Architecture verstehen
- DTOs als Records gestalten - niemals Domain-Objekte exponieren
- Mapping zwischen DTOs, Commands und Domain sicher umsetzen
- REST-Endpunkte für CRUD korrekt mit HTTP-Statuscodes modellieren
- Problem Details (RFC 9457) mit Spring Boot 4 implementieren
- Integration Tests mit `@WebMvcTest` schreiben

Der REST Adapter in Clean Architecture



- Der Controller ist ein Adapter - er übersetzt HTTP in Domain-Sprache
- Er kennt `application.port`, aber nicht `infrastructure`

Warum DTOs? Niemals Domain-Objekte exponieren

Problem	Was passiert ohne DTOs
Kopplung	API-Änderung erzwingt Domain-Änderung (und umgekehrt)
Sicherheit	Interne Felder werden sichtbar (z.B. `domainEvents`)
Serialisierung	Jackson-Annotationen wandern in die Domain
Versionierung	Keine unabhängige API-Evolution möglich
Zirkuläre Deps	Domain kennt plötzlich `com.fasterxml.jackson`

```
// NEVER:
@GetMapping("/{id}")
public BrokerageProcess getById(@PathVariable UUID id) {
    return repository.findById(new ProcessId(id)).orElseThrow();
}
```

> DTOs gehören zum Adapter - sie sind der API-Kontrakt mit der Außenwelt.

Record-basierte DTOs - Request

```
package de.realestate.brokerage.adapter.web;

public record CreateViewingRequest(
    @NotNull UUID processId,
    @NotNull @Future LocalDateTime appointmentDate,
    @NotNull UUID prospectId
) {
    // Mapping: DTO → Command (primitive types → value objects)
    public ScheduleViewingCommand toCommand() {
        return new ScheduleViewingCommand(
            new ProcessId(processId),
            new ContactId(prospectId),
            appointmentDate);
    }
}
```

Record-basierte DTOs - Request

- Java Records sind ideal für DTOs: immutable, kompakt
- Bean Validation direkt auf den Record-Komponenten
- `toCommand()` erzeugt Value Objects aus primitiven Typen
- Kein Boilerplate: `equals()`, `hashCode()`, `toString()` inklusive
- `toCommand()` liegt bewusst im DTO: der Adapter darf den Application-Layer kennen

Record-basierte DTOs - Response

```
public record ViewingResponse(
    UUID viewingId,
    UUID processId,
    LocalDateTime appointmentDate,
    String status
) {
    // Factory: domain result → response DTO
    public static ViewingResponse from(
            ViewingId id,
            ProcessId processId,
            LocalDateTime appointmentDate,
            ViewingStatus status) {
        return new ViewingResponse(
            id.value(), processId.value(), appointmentDate, status.name());
    }
}
```

Record-basierte DTOs - Response

```
public record ProcessDetailResponse(
    UUID id, String status, int viewingCount,
    List<ViewingSummaryResponse> viewings
) {
    public static ProcessDetailResponse from(ProcessDetails details) { /* ... */ }
}
```

- Response-DTOs verwenden primitive Typen (UUID, String) - keine Value Objects
- `from()`-Factory macht das Mapping explizit und testbar

REST Controller - POST (Ressource erstellen)

```
@RestController
@RequestMapping("/api/v1/brokerage/viewings")
public class ViewingController {

    private final ScheduleViewing scheduleUseCase;
    private final QueryViewings queryUseCase;
    private final CompleteViewing completeUseCase;

    public ViewingController(ScheduleViewing scheduleUseCase,
                             QueryViewings queryUseCase,
                             CompleteViewing completeUseCase) {
        this.scheduleUseCase = scheduleUseCase;
        this.queryUseCase = queryUseCase;
        this.completeUseCase = completeUseCase;
    }

    @PostMapping
    public ResponseEntity<ViewingResponse> create(
            @Valid @RequestBody CreateViewingRequest request) {
        var result = scheduleUseCase.schedule(request.toCommand());
        var uri = ServletUriComponentsBuilder.fromCurrentRequest()
            .path("/{id}").buildAndExpand(result.id().value()).toUri();
        var response = ViewingResponse.from(
            result.id(), result.processId(),
            result.appointmentDate(), result.status());
        return ResponseEntity.created(uri).body(response);
    }
}
```

> `ServletUriComponentsBuilder` erzeugt die `Location`-URI inkl. Host, Port und Context-Path.

REST Controller - GET und PUT

```
@GetMapping("/{id}")
public ResponseEntity<ViewingResponse> getById(
        @PathVariable UUID id) {
    return queryUseCase.findById(new ViewingId(id))
        .map(ViewingResponse::from)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}

@PutMapping("/{id}/complete")
public ResponseEntity<Void> complete(@PathVariable UUID id) {
    completeUseCase.execute(
        new CompleteViewingCommand(new ViewingId(id)));
    return ResponseEntity.noContent().build();
}
```

- Alle Use Cases sind im Konstruktor deklariert (vorherige Slide)
- GET gibt `ViewingResponse` zurück - konsistent mit dem Ressourcen-Typ
- PUT auf Sub-Ressource `/complete` modelliert eine fachliche Aktion

HTTP Status Codes - Best Practices

Methode	Erfolg	Fehler
`POST` (erstellen)	`201 Created` + `Location`-Header	`400` / `422`
`GET` (lesen)	`200 OK`	`404 Not Found`
`PUT` (ändern)	`200 OK` oder `204 No Content`	`404` / `422`
`DELETE` (löschen)	`204 No Content`	`404`

Fachliche Fehler vs. technische Fehler

Kategorie	HTTP-Status	Beispiel
Syntaktisch ungültig	`400 Bad Request`	Bean Validation fehlgeschlagen
Ressource nicht gefunden	`404 Not Found`	Unbekannte ProcessId
Fachliche Regel verletzt	`422 Unprocessable Entity`	Max. Besichtigungen erreicht
Interner Fehler	`500 Internal Server Error`	Unerwarteter Datenbankfehler

Problem Details - RFC 9457

Spring Boot 4 unterstützt RFC 9457 nativ mit der `ProblemDetail`-Klasse:

```
{
  "type": "https://api.immo-crm.de/errors/process-not-found",
  "title": "BrokerageProcess not found",
  "status": 404,
  "detail": "No process with ID 550e8400-e29b-41d4-a716-446655440000",
  "instance": "/api/v1/brokerage/viewings"
}
```

Aktivierung

```
# application.yml
spring:
  mvc:
    problemdetails:
      enabled: true
```

- Standardisiertes Fehlerformat - maschinenlesbar und menschenlesbar
- Jeder Fehlertyp bekommt eine eigene URI (`type`-Feld)

@RestControllerAdvice mit ProblemDetail

```
@RestControllerAdvice
public class DomainExceptionHandler {

    @ExceptionHandler(ProcessNotFoundException.class)
    public ProblemDetail handleNotFound(ProcessNotFoundException ex) {
        var problem = ProblemDetail.forStatusAndDetail(
            HttpStatus.NOT_FOUND, ex.getMessage());
        problem.setTitle("BrokerageProcess not found");
        problem.setType(URI.create(
            "https://api.immo-crm.de/errors/process-not-found"));
        return problem;
    }

    @ExceptionHandler(DomainException.class)
    public ProblemDetail handleDomainViolation(DomainException ex) {
        var problem = ProblemDetail.forStatusAndDetail(
            HttpStatus.UNPROCESSABLE_ENTITY, ex.getMessage());
        problem.setTitle("Domain rule violated");
        problem.setType(URI.create(
            "https://api.immo-crm.de/errors/domain-violation"));
        return problem;
    }
}
```

> Alternative: Für einfache Fälle ohne eigene `type`-URI reicht `@ResponseStatus(HttpStatus.NOT_FOUND)` direkt auf der Exception-Klasse. `@RestControllerAdvice` lohnt sich, wenn `ProblemDetail`-Felder individuell gesetzt werden sollen.

Integration Test mit @WebMvcTest

```
@WebMvcTest(ViewingController.class)
class ViewingControllerTest {

    @Autowired private MockMvc mockMvc;
    @MockitoBean private ScheduleViewing scheduleUseCase;
    @MockitoBean private QueryViewings queryUseCase;
    @MockitoBean private CompleteViewing completeUseCase;

    @Test
    void should_create_viewing() throws Exception {
        var result = new ScheduleViewingResult(
            new ViewingId(UUID.randomUUID()),
            new ProcessId(UUID.randomUUID()),
            LocalDateTime.of(2026, 4, 15, 14, 0),
            ViewingStatus.SCHEDULED);
        when(scheduleUseCase.schedule(any())).thenReturn(result);

        mockMvc.perform(post("/api/v1/brokerage/viewings")
                .contentType(MediaType.APPLICATION_JSON)
                .content("""
                    {
                      "processId": "550e8400-e29b-41d4-a716-446655440000",
                      "prospectId": "660e8400-e29b-41d4-a716-446655440000",
                      "appointmentDate": "2026-04-15T14:00:00"
                    }
                    """))
            .andExpect(status().isCreated())
            .andExpect(header().exists("Location"))
            .andExpect(jsonPath("$.viewingId").value(
                result.id().value().toString()));
    }
}
```

@WebMvcTest - Fehlerfall testen

```
@Test
void should_return_404_when_process_does_not_exist() throws Exception {
    when(scheduleUseCase.schedule(any()))
        .thenThrow(new ProcessNotFoundException(
            new ProcessId(UUID.randomUUID())));

    mockMvc.perform(post("/api/v1/brokerage/viewings")
            .contentType(MediaType.APPLICATION_JSON)
            .content("""
                {
                  "processId": "550e8400-e29b-41d4-a716-446655440000",
                  "prospectId": "660e8400-e29b-41d4-a716-446655440000",
                  "appointmentDate": "2026-04-15T14:00:00"
                }
                """))
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("$.title")
            .value("BrokerageProcess not found"));
}
```

- `@WebMvcTest` lädt einen reduzierten Spring-Kontext - nur Web-Layer, keine Services, keine DB
- Use Cases werden mit `@MockitoBean` gemockt
- Testet: Routing, Serialisierung, Validation, Exception Handling

Zusammenfassung

- `@RestController` ist ein Inbound Adapter - übersetzt HTTP ↔️ Domain-Sprache
- DTOs als Records - niemals Domain-Objekte über die API exponieren
- Request-DTO → `toCommand()` → Value Objects, Result → `from()` → Response-DTO
- HTTP Status Codes: `201` (POST), `200` (GET), `204` (PUT/DELETE), `404`, `422`
- Problem Details (RFC 9457) für standardisierte Fehlermeldungen
- `@RestControllerAdvice` fängt Domain Exceptions und mappt auf `ProblemDetail`
- `@WebMvcTest` testet den Adapter isoliert ohne Datenbank

Hands-on: Lab 07

REST-Adapter implementieren

1. Request- und Response-DTOs als Java Records erstellen
2. `ViewingController` mit POST und GET Endpunkt implementieren
3. Manuelles Mapping: `toCommand()` und `from()` Methoden
4. `@RestControllerAdvice` mit Problem Details (RFC 9457) einrichten
5. Integration Test mit `@WebMvcTest` schreiben

> Dauer: ca. 45 Minuten
> Details und Aufgabenstellung im Lab-07

Diskussion

> Welche API-Design-Entscheidungen sind in eurem Kontext wichtig?

- Wie handhabt ihr API-Versionierung (URL-Pfad, Header, Query-Parameter)?
- Wie dokumentiert ihr eure APIs (OpenAPI/Swagger, Spring REST Docs)?