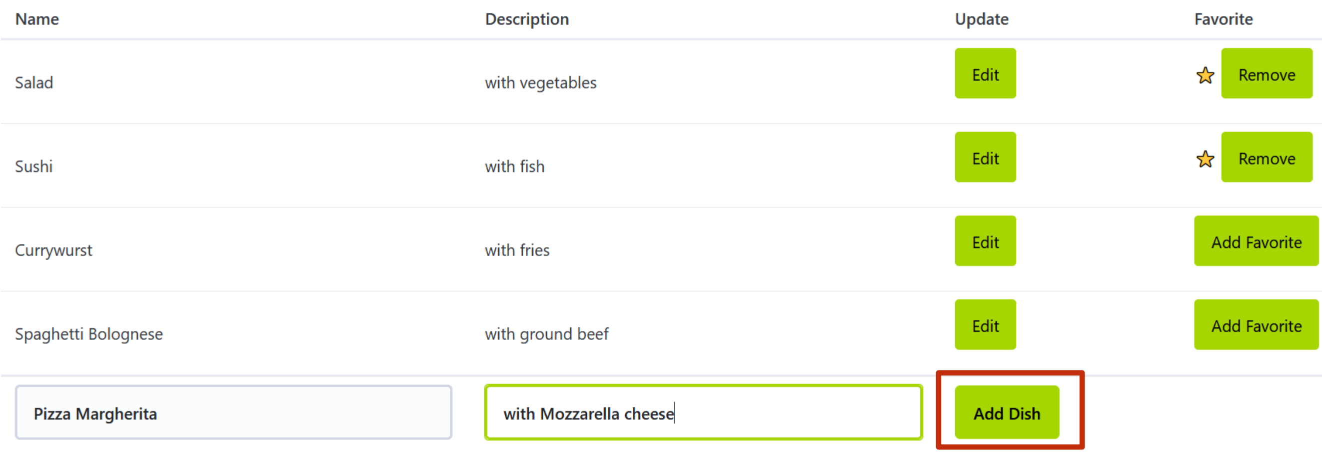
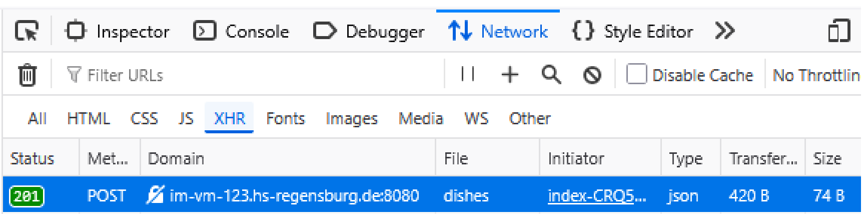
02 – Backend Development

Web Technology Project (International Computer Science)
Summer semester 2025
Prof. Dr. Felix Schwägerl

OTH REGENSBURG

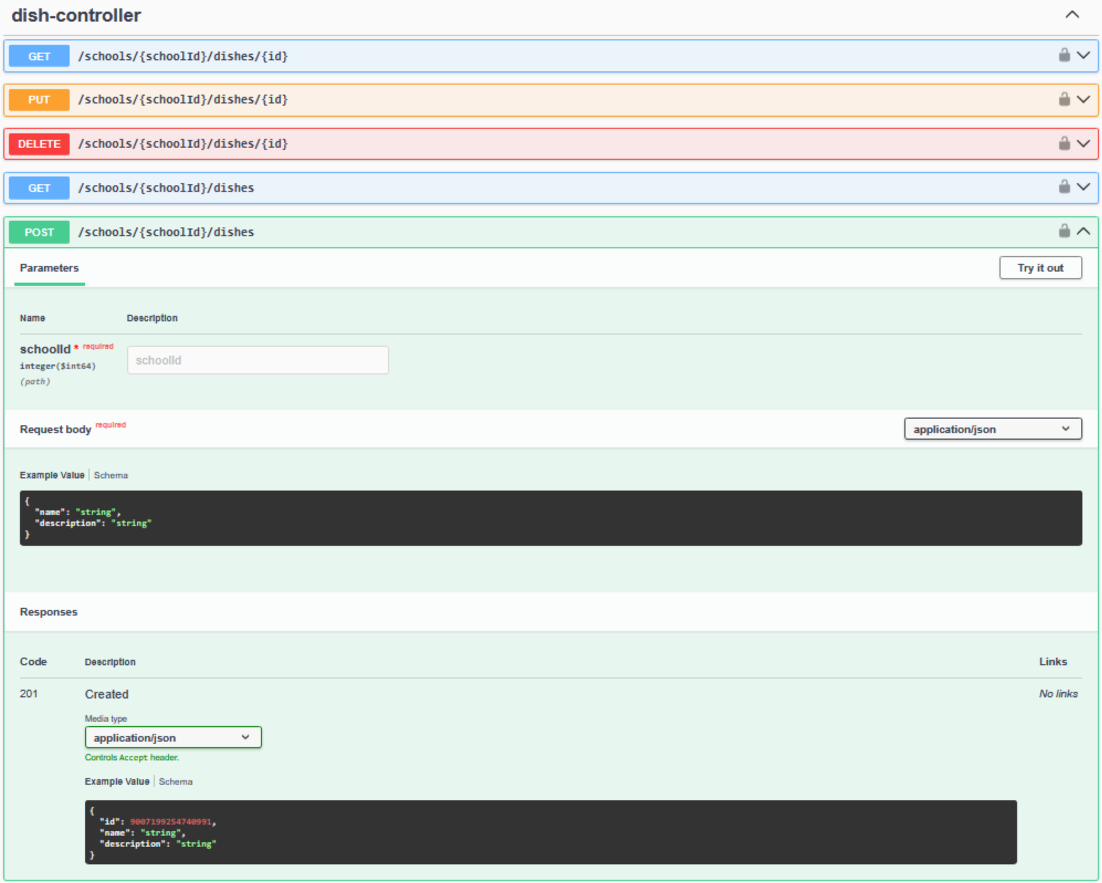
Mensa app: Example request and response





- Request body: `{"name":"Pizza Margherita","description":"with Mozzarella cheese"}`
- Response body: `{"id":52,"name":"Pizza Margherita","description":"with …"}`

Mensa app: Swagger/OpenAPI documentation



[http://im-vm-123.hs-regensburg.de:8080/swagger-ui/index.html#/dish-controller/createDish]

Backend web frameworks: Brief historical overview

- **Early Java Web Development (1990s):** *Servlets* and JSP (JavaServer Pages), providing basic web capabilities through low-level API interactions with HTTP requests and responses.
- **J2EE (Java 2 Platform, Enterprise Edition)**: In the late 1990s, J2EE emerged, standardizing enterprise applications with components like EJB (Enterprise JavaBeans), Servlets, and JSP.
- **Struts (2000)**: Apache Struts, one of the first major MVC (Model-View-Controller) frameworks, became popular, providing a structure for building maintainable web applications in Java.
- **Spring Framework (2002)**: The Spring Framework introduced a lightweight, modular approach to Java web development, emphasizing dependency injection, aspect-oriented programming, and easier integration with other technologies, making it a go-to framework for Java developers.
- **Java Server Faces (JSF, 2004)**: JSF, part of the J2EE standard, brought a component-based approach to building web UIs, although criticized for its complexity and steep learning curve.
- **The Rise of REST APIs (2000s)**: With the growing demand for mobile and web-based client-server communication, RESTful web services became a key focus, encouraging frameworks like Spring MVC and JAX-RS to support REST API development.
- **Spring Boot (2013)**: Spring Boot revolutionized Java web development by providing an opinionated, convention-over-configuration approach, making it easier to develop production-ready web applications with minimal setup and boilerplate code.
- **Microservices and Cloud-Native Development (2010s–Present)**: The advent of microservices architecture and the rise of cloud computing led to frameworks likeSpring Cloud, which provide tools for building scalable, distributed, and cloud-native Java applications.

[ChatGPT1]

Spring

- **Lightweight & Modular**: Spring is a comprehensive framework that focuses on flexibility and modularity. It allows developers to use only the components they need.
- **Persistence Support with JPA/Hibernate**: Spring simplifies database interaction through its support for *JPA (Java Persistence API)* and *Hibernate*, making it easier to manage database transactions and entities.
- **RESTful APIs**: Spring supports *JAX-RS (Jakarta RESTful Web Services)* for building RESTful APIs, enabling easy integration between web services and applications
- **Integration with Tomcat**: Spring can be integrated with web servers like Tomcat or Jetty for handling HTTP requests, providing a seamless environment for web applications.

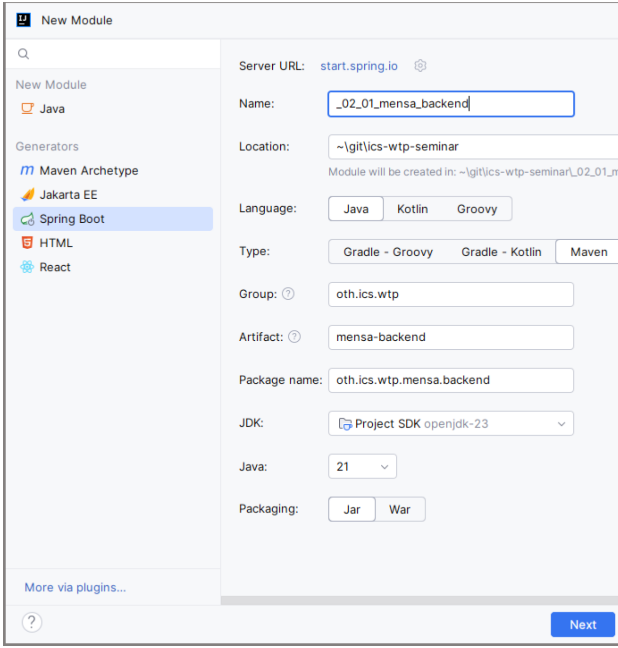
Spring Boot

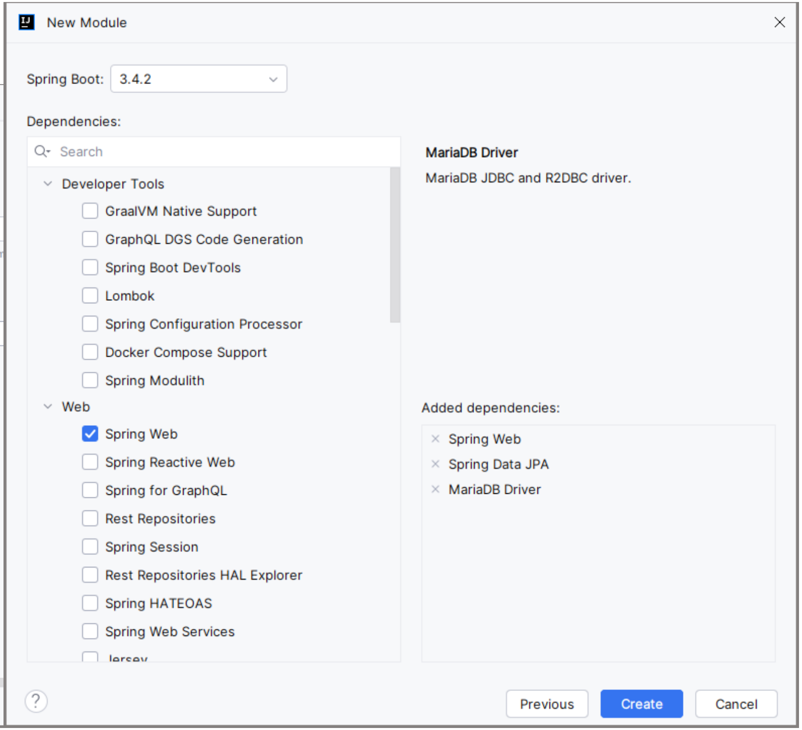
- **Opinionated & Convention-over-Configuration**: Spring Boot simplifies the setup and configuration of Spring applications by providing sensible defaults, reducing the need for manual configuration.
- **Embedded Server:** Spring Boot comes with an embedded *Tomcat*, allowing applications to be run as standalone applications without needing an external servlet container.
- **Auto Configuration**: Spring Boot automatically configures various components like JPA, security, and database connections, streamlining application setup.

[ChatGPT2]

Spring initializr

- Web version: https://start.spring.io/ (generates a downloadable archive)
- Integrated into IntelliJ:





The generated Maven build file (pom.xml)

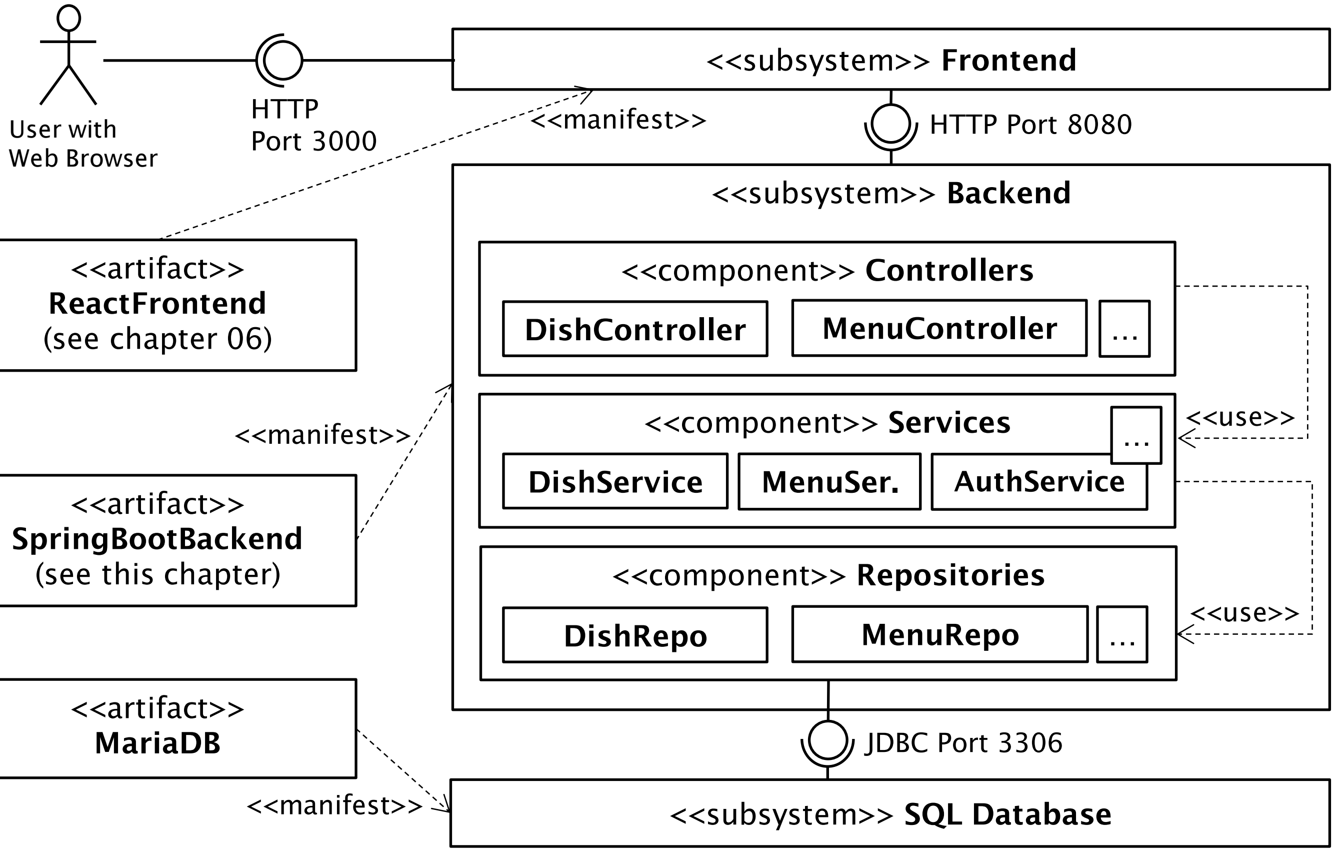
```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>oth.ics.wtp</groupId>
    <artifactId>mensa-backend</artifactId>
    <version>1</version>
    <name>_02_01_mensa_backend</name>
    <description>_02_01_mensa_backend</description>
    <properties>
        <java.version>21</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Basic properties are based on parameters provided for project creation.

Spring Boot can be upgraded by incrementing version,

Dependencies selected in initializr (will be changed afterwards, e.g., to add test-dependencies to H2 database and support for Swagger/OpenAPI)

Reference architecture assumed in WTP



Controller

- Receives HTTP requests and their parameters.
- Converts between JSON request/response bodies and *data transfer objects* (DTOs).
- Each controller describes the CRUD operation of a domain concept (e.g., school, dish, menu).
- A controller implementation may refer to multiple *services*.

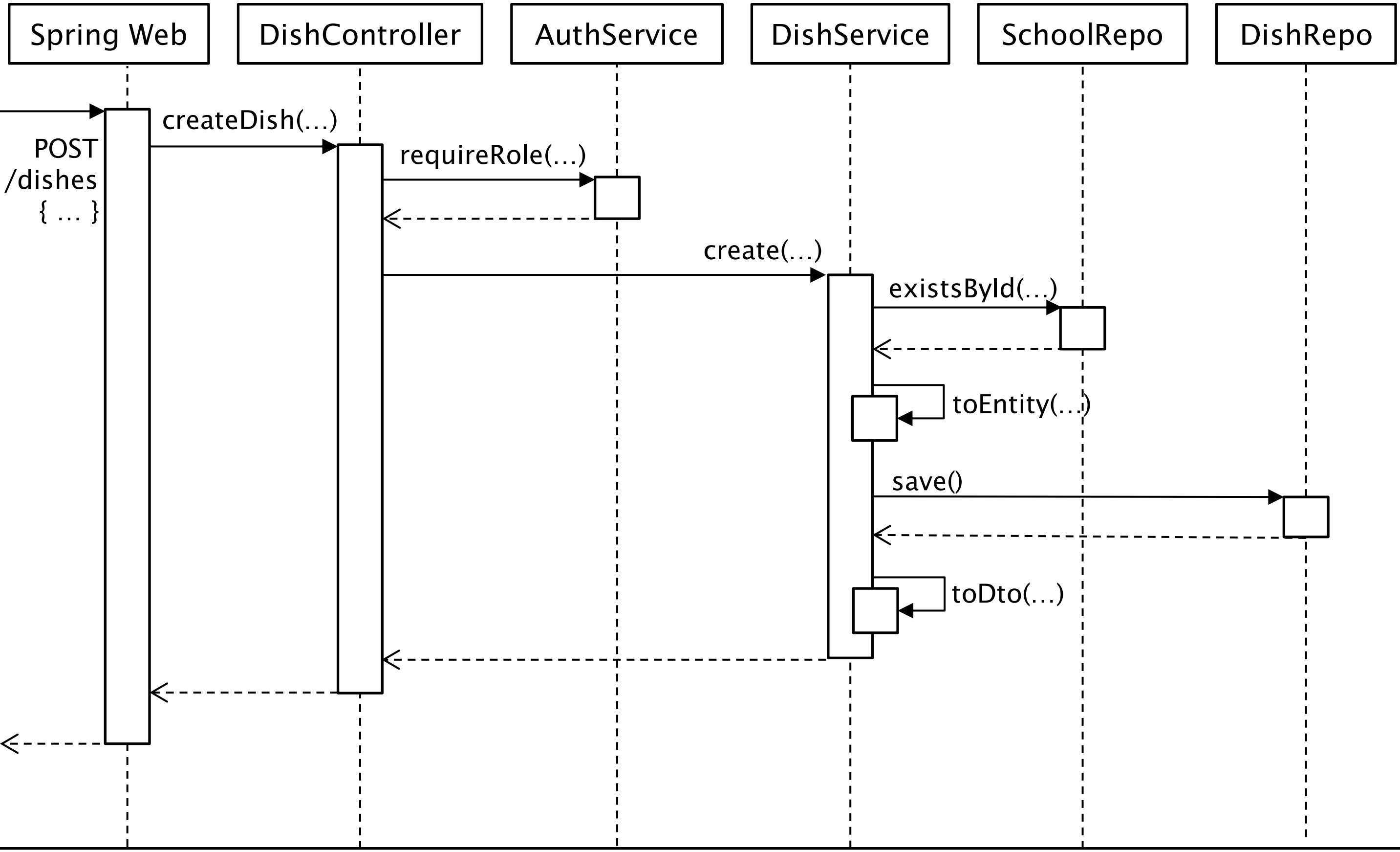
Service

- Offers operations based on DTOs
- Converts between *DTO* and *entity* representation.
- Manages *sessions*, *authentication*, and *error* handling.
- A service implementation may refer to multiple *repositories*.

Repository

- Abstraction layer for persistence *entities* (using Java Persistence Framework)
- Every repository corresponds to *a SQL database relation/table.*
- Every repositoy stores a specific kind of *entity*.
- Spring Boot offers a powerful *query* language, making manual SQL queries unnecessary.

Example request processing



Spring enforces loosely coupled components by DI

Definition of a component (here, a service of the middle layer):

```
@Service public class DishService { … }
```

Usage of components (here, a controller of the top layer):

```
@RestController public class DishController {
    private final AuthService authService;
    private final DishService dishService;

    @Autowired public DishController(AuthService as, DishService ds) {
        this.authService = as;
        this.dishService = ds;
    } …
}
```

- DI style preferred here: *constructor injection*. Alternatives: *field injection* (not recommended), *setter injection* (necessary in case of cyclic dependencies)
- Default resolution: DI creates instance of the used class
  - More complex resolution can be realized, e.g., allowing for run-time (re-)configurability or for mocking components in tests
- Never directly use constructors across layers! Use DI/`@Autowired` instead!

02_01_mensa_backend/src/…/
DishService.java
DishController.java

Spring Boot allows fine-grained configuration, while assuming reasonable defaults (convention over configuration).

```
spring.application.name=mensa_backend
spring.datasource.url=jdbc:mariadb://127.0.0.1:3306/mensa?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=asdf1234
spring.jpa.hibernate.ddl-auto=update
logging.level.org.springframework.web.filter.CommonsRequestLoggingFilter=debug
```

- Datasource URL: local MariaDB deployment (see next slide)
  - URL may also refer to an external database (e.g., personal OTH PostgreSQL)
  - JPA normally expects a database with specified name to pre-exist.
  - Query option `createDatabaseIfNotExist=true` automatically creates the DB on first run.
- Datasource username/password: All requests here use the same credentials. (defined on the deployment as shown on next slide)
- DDL auto update allows to re-execute the application after a schema change (normally, a complex migration is required)
- Last line logs all REST requests to the console (useful for debugging)

Docker-compose config for local test database (MariaDB)

```yaml
services:
  db:
    image: mariadb
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: asdf1234
    volumes:
      - ./db-data:/var/lib/mysql
    ports:
      - 3306:3306
  adminer:
    image: adminer
    restart: always
    ports:
      - 7070:8080
    depends_on:
      - db
```

MariaDB server with persistent storage in the folder `db-data` (relative to parent folder of `docker-compose.yml`).

SQL interface is exposed to port 3306.

Optional DB administration interface allowing to access the database contents (useful for debugging).

Web interface is exposed to port 7070.

- To start the database, go to the folder containing the `docker-compose.yml`.
- Then execute in shell (e.g., Linux or Powershell): `docker-compose up -d`
- The database data is persisted locally; the service is restarted upon reboot.
- Adminer interface is available via `http://localhost:7070`

Bottom layer: Repositories

- oth.ics.wtp.mensa.backend.repositories
  - AppUserRepository
  - CommentRepository
  - DishRepository
  - FavoriteRepository
  - MenuRepository
  - SchoolRepository

Entity type (see next slide)

Primary key type

```java
public interface DishRepository extends CrudRepository<Dish, Long> {
    List<Dish> findBySchoolId(long schoolId);
    Optional<Dish> findBySchoolIdAndId(long schoolId, long id);
    boolean existsBySchoolIdAndId(long schoolId, long id);
}
```

- In most cases, repositories do not require manual implementation
  - Merely an *interface* needs to be extended.
- Objects of the entity type are automatically transformed from/to SQL rows.
- SQL queries are automatically derived from the names of method, e.g.
  - `findBySchoolIdAndId` → **WHERE** `school.id = schoolId` **AND** `id = id`
- https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

Persistence entities

oth.ics.wtp.mensa.backend.entities
- AppUser
- Comment
- Dish
- Favorite
- Menu
- School

```
@Entity public class Dish {
    @Id @GeneratedValue private long id;
    private String name;
    private String description;

    public Dish() { }
    public Dish(String name, String description, School school) {
        this.name = name;
        this.description = description;
        this.school = school;
    }

    public long getId() {…}
    public String getName() {…}
    public String getDescription() {…}
    public void setName(String name) {…}
    public void setDescription(String description) {…}
}
```

JPA Entity annotation

Primary key with auto-increment value

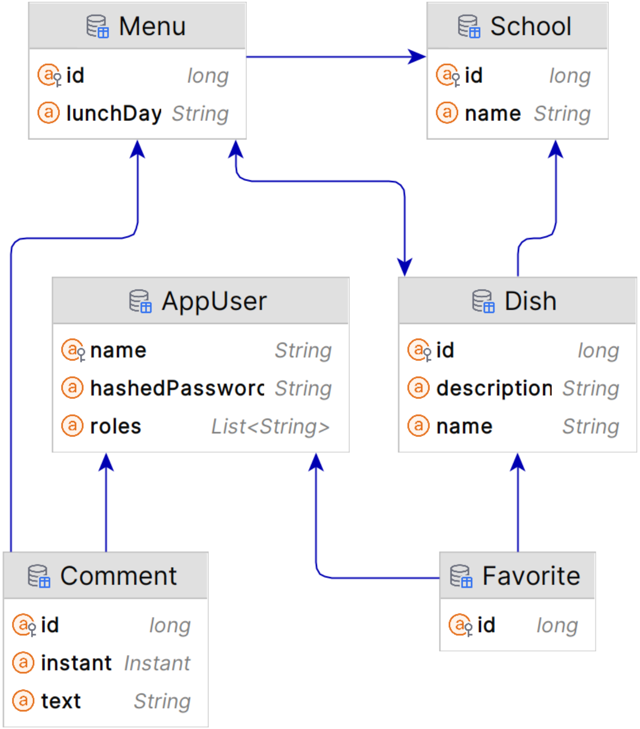
Non-key columns

Required default constructor

- Java attributes are mapped to SQL columns in a straightforward wa
  - Data types are converted, e.g., `String` into VARCHAR
- JPA requires a default constructor.
  - Non-default constructors, getters and setters may be added as required.

IntelliJ may generate an ER diagram from the current entities

- Go to persistence view, right-click on root and choose *Entity Relationship Daigram*
- Limitations: Java types (not SQL compliant), no cardinalities



Relationships between entities are implemented via SQL foreign keys.

- Example: Uni-directional single-valued relationship from `Dish` to `School`

```
@Entity public class Dish {
    @Id @GeneratedValue private long id;
    private String name;
    private String description;
    @ManyToOne @OnDelete(action = CASCADE) private School school;
    ...
}
```

Many dishes have one school (M:1)

If a school is deleted, all its dishes are deleted.

Reference type of foreign key attribute to be generated

- Type of a ManyToOne annotated field must be another entity.
- Here, table `Dish` receives a foreign key attribute to `school.id`

Many-valued relationships are represented as Java collections.

- Example: Bi-directional multi-valued relationship from `Menu` to `Dish`.

```java
@Entity public class Menu {
    @Id @GeneratedValue private long id;
    private String lunchDay;
    @ManyToOne @OnDelete(action = CASCADE) private School school;
    @ManyToMany(fetch = EAGER) private List<Dish> dishes;
}
```

Many menus contain many dishes (M:N)

When retrieving a menu from the database, its dishes should be retrieved, too.

- Here, Java collection type `List` is used (duplicates are allowed).
- In the background, an additional table is generated for M:N relationships.
  - This is entirely transparent to Spring Boot.
  - Adminer view:

```sql
SELECT * FROM `menu_dishes` LIMIT 50
```
(0.000 s) Edit

Modify	menus_id	dishes_id
edit	1	1
edit	1	2
edit	2	3

Persistent relationships are navigable in reverse direction.

- Example: The menus in which a specific dish appears.
  - Perstently stored at the opposite side (`Menu.dishes`)
  - JPA allows to read and manipulate via reverse direction using the `mappedBy` annotation attribute.

Not stored persistently, but retrieved from opposite entity

```
@Entity public class Dish {
    ...
    @ManyToMany(mappedBy = "dishes") private List<Menu> menus;
    ...
    @PreRemove private void onDelete() {
        for (Menu menu : menus) {
            menu.getDishes().remove(this);
        }
    }
}
```

Manual implementation of ON DELETE CASCADE behavior

- When a dish is deleted, it is supposed to be deleted from all menus, too.
- For many-to-many relationships, the ON DELETE CASCADE behavior has to be implemented manually.
  - Here solved using the `PreRemove` hook `onDelete`.

What about multi-valued attributes?

- Normal form of relational model does not allow multiple values for one column.
- Having to create a dedicated entity for nested data would be cumbersome.
- Solution: *element collections*
  - Externally treated like collections of primitive values.
  - Internally mapped to additional SQL relations/tables (not exposed to Spring Boot)

```
@Entity public class AppUser {
    @Id private String name;
    private String hashedPassword;
    @ElementCollection(fetch=EAGER) @CollectionTable private List<String> roles;
    …
}
```

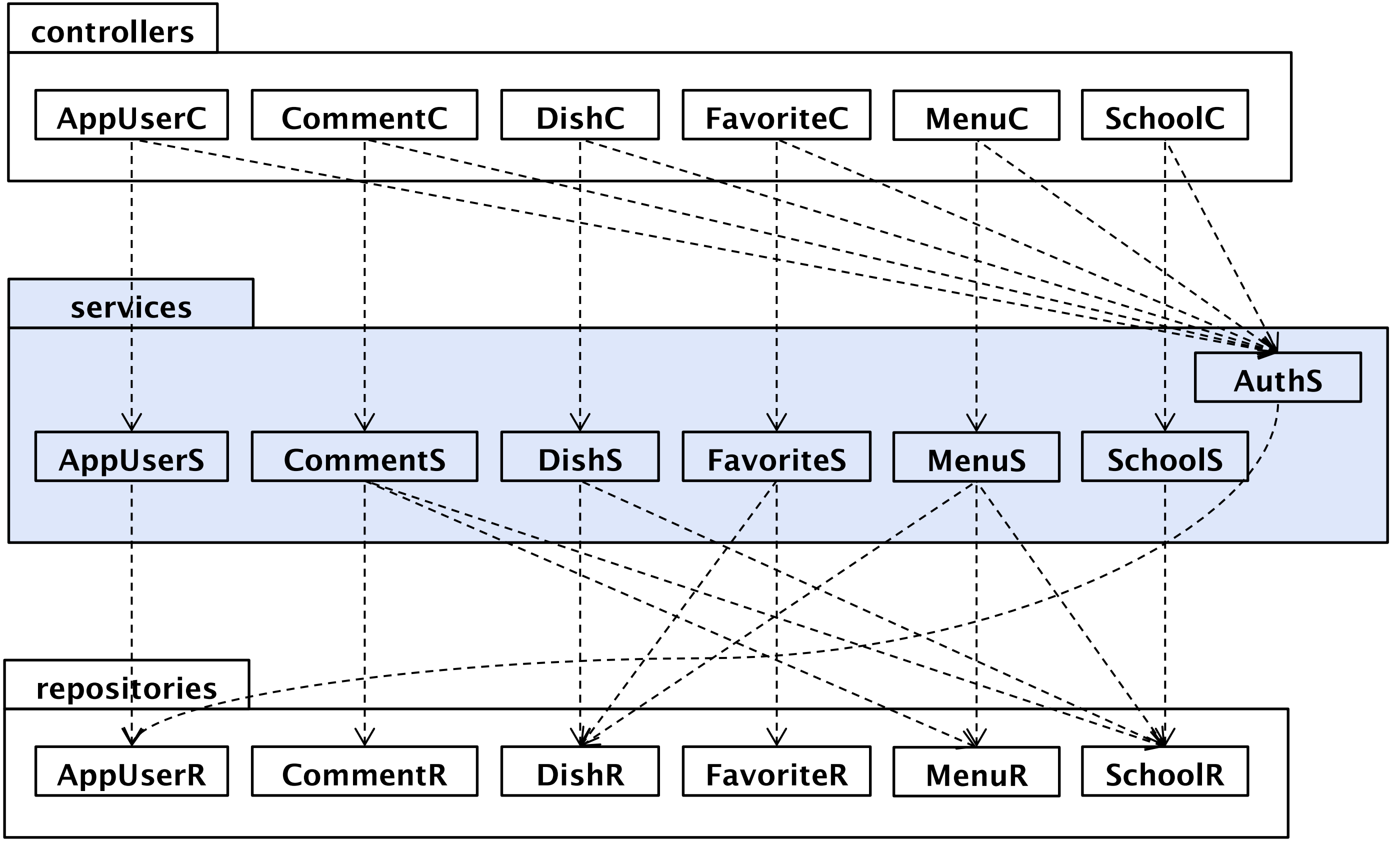
When retrieving a user from the database, roles are retrieved, too.

This is implemented by a transparent additional SQL table.

- Adminer view:

SELECT * FROM `app_user_roles` LIMIT 50 (0.000 s) Edit

Modify	app_user_name	roles
edit	admin	USER
edit	admin	MANAGER
edit	admin	ADMIN
edit	felix	USER


controllers
AppUserC
CommentC
DishC
FavoriteC
MenuC
SchoolC
services
AuthS
AppUserS
CommentS
DishS
FavoriteS
MenuS
SchoolS
repositories
AppUserR
CommentR
DishR
FavoriteR
MenuR
SchoolR

Data transfer objects (DTOs)

- oth.ics.wtp.mensa.backend.dtos
  - AppUserDto
  - CommentDto
  - CreateAppUserDto
  - CreateCommentDto
  - CreateDishDto
  - CreateSchoolDto
  - DishDto
  - MenuDto
  - SchoolDto
  - UpdateDishDto
  - UpdateDishOfMenuDto
  - UpdateMenuDto

```java
public record CreateDishDto(
    String name, String description) { }
```

```java
public record DishDto(long id,
    String name, String description) { }
```

```java
public record UpdateDishDto(
    String name, String description) { }
```

```java
public record UpdateDishOfMenuDto(long id) { }
```

- While entities reflect the internal persistent data schema, DTOs should reflect structured data as it is *read* and *written* by *API* users by *CRUD operations*.
- One entity may be reflected by multiple DTOs, each considering a different *intention* (relevant to a specific *use case*) of it. → *Single Responsibility Principle*
- Here, DTOs are mapped to JSON 1:1.
- Since DTOs are *immutable*, Java *records* are a concise and safe choice to implement them.

02_01_mensa_backend/…/dtos/
CreateDishDto.java DishDto.java
UpdateDishDto.java
UpdateDishOfMenuDto.java

Services *adapt* DTOs to entities.

- External interface (used by controllers): DTOs
- Internal implementation detail (used by repositories): entities

```
@Service public class DishService {
    …

    private DishDto toDto(Dish dish) {
        return new DishDto(dish.getId(), dish.getName(), dish.getDescription());
    }

    private Dish toEntity(School school, CreateDishDto createDish) {
        return new Dish(createDish.name(), createDish.description(), school);
    }
}
```

Used for getting
(see next slide)

Used for inserting
(see next slide)

Services offer domain-oriented access operations.

- Consistent and minimal units of behavior, implemented using repositories

```
@Service public class DishService {
    private final SchoolRepository schoolRepo;
    private final DishRepository dishRepo;
    @Autowired public DishService(SchoolRepository sr, DishRepository dr) {
        this.schoolRepo = sr; this.dishRepo = dr;
    }

    public DishDto create(long schoolId, CreateDishDto createDish) {
        School school = schoolRepo.findById(schoolId)
                                  .orElseThrow(() -> ClientErrors.schoolNotFound(schoolId));
        Dish entity = toEntity(school, createDish);
        dishRepo.save(entity);
        return toDto(entity);
    }

    public DishDto get(long schoolId, long id) {
        if (!schoolRepo.existsById(schoolId)) {
            throw ClientErrors.schoolNotFound(schoolId);
        }
        return dishRepo.findBySchoolIdAndId(schoolId, id)
                .map(this::toDto)
                .orElseThrow(() -> ClientErrors.dishNotFound(id));
    }
}
```

Exceptions must be translated into HTTP responses.

- Default behavior if any exception is thrown: 500 (Internal server error)
- Many errors are due to *client errors* (e.g., invalid IDs specified).
  - In such a case, the error needs to be reported to the client with adequate detail.
- Spring contains a specific exception class, `ResponseStatusException`, which is configuratble with respect to HTTP *status code* and *message*.
- The Mensa example collects frequent error types in a class `ClientErrors`.
  - In addition to returning the error via HTTP, it is logged to the console.

```java
public class ClientErrors {
    private static final Logger logger = LoggerFactory.getLogger(ClientErrors.class);
    public static ResponseStatusException forbidden(String role) {
        return log(new ResponseStatusException(HttpStatus.FORBIDDEN, "missing role: " + role));
    }
    public static ResponseStatusException schoolNotFound(long id) {
        return log(new ResponseStatusException(HttpStatus.NOT_FOUND, "school with id " + id));
    }
    public static ResponseStatusException userNameTaken(String name) {
        return log(new ResponseStatusException(HttpStatus.BAD_REQUEST, "name already taken: "+name));
    }
    ...
    private static ResponseStatusException log(ResponseStatusException e) {
        logger.error(ExceptionUtils.getMessage(e) + "\n" + ExceptionUtils.getStackTrace(e));
        return e;
    }
}
```

Controllers are the entry points of the API.

- They offer REST endpoints via annotated Java methods.
- Spring Boot automatically converts between *JSON* and *DTO classes*.
- Following our reference architecture, controllers should have minimal implementation and delegate as much work as possible to *services*.

```java
@RestController public class DishController {
    private final AuthService authService;
    private final DishService dishService;

    @Autowired public DishController(AuthService as, DishService ds) {
        this.authService = as; this.dishService = ds;
    }

    @GetMapping(value = "schools/{schoolId}/dishes")
    public List<DishDto> listDishes(HttpServletRequest request,
            @PathVariable("schoolId") long schoolId) {
        authService.requireRole(request, Roles.USER);
        return dishService.list(schoolId);
    }
}
```

Controller depends on two services

Method handles GET request for a given REST path

Parameter is extracted from variable in REST path

Low-level request object is captured

Delegating to services

Result list is converted into JSON array

All parts of a REST request may be extracted as parameters.

- Abstract example request:

```
POST /things/23/foo?qp=bar
Header1: value1

{ bodyValue: 42 }
```

Request may contain information in its different parts: path, query, header, body

- DTO for capturing body:

```java
public record BodyDto(int bodyValue) { }
```

- Controller method:

```java
@PostMapping(value = „things/{thingId}/foo")
public BodyDto postFoo(HttpServletRequest request,
        @PathVariable("thingId") long thingId,
        @RequestParam("qp") String qp,
        @RequestHeader("Header1") String header1,
        @RequestBody BodyDto body) {
    System.out.println(body.bodyValue());
    …
    return body;
}
```

Low-level request object allows to access all request info dynamically (but less user friendly)

Specific annotations allow to extract request information in a fine-grained way

Response body is constructed from returned value (default: JSON)

Annotations control the properties of the response.

```
@RestController public class DishController {
    …
    @GetMapping(value = "schools/{schoolId}/dishes",
                produces = MediaType.APPLICATION_JSON_VALUE)
    public List<DishDto> listDishes(…) {
        …
    }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping(value = "schools/{schoolId}/dishes",
                 consumes = MediaType.APPLICATION_JSON_VALUE,
                 produces = MediaType.APPLICATION_JSON_VALUE)
    public DishDto createDish(…, @RequestBody CreateDishDto createDish) {
        …
    }
}
```

Represent response body as application/json

Return specific status code (201)

Expect request body as application/json

- Default: status code 200, content type `application/json` (with exceptions to the rule)
- Alternatively, it is also possible to capture a low-level response object `HttpServletResponse` and return this. (E.g., when specific headers must be set.)

App users are covered by the three regular layers.

```
@Entity public class AppUser {
    @Id private String name;
    private String hashedPassword;
    @ElementCollection(fetch = EAGER) @CollectionTable private List<String> roles;
    ...
}

public interface AppUserRepository extends CrudRepository<AppUser, String> {
    Optional<AppUser> findByName(String username);
    boolean existsByName(String name);
}

public record AppUserDto(String name, List<String> roles) { }

public record CreateAppUserDto(String name, String password, List<String> roles) { }

@Service public class AppUserService { … }

@RestController public class AppUserController { …
    @PostMapping(value = "users/login")
    public AppUserDto logIn(HttpServletRequest request) {
        AppUser user = authService.logIn(request);
        return userService.get(user.getName());
    }
}
```

(Weak) security: Store MD5 hash instead of clear-text password

API for creating, getting, listing, deleting users

Log-in endpoint that expects basic auth in header

App users are also the foundation of *authentication* and *authorization*.

```java
@Service public class AuthService {
    public void requireRole(AppUser user, String role) {
        if (role != null && !role.isEmpty() && !user.getRoles().contains(role)) {
            throw ClientErrors.forbidden(role);
        }
    }

    public AppUser logIn(HttpServletRequest request) {
        try {
            String ah = request.getHeader(HttpHeaders.AUTHORIZATION);
            String decoded = WeakCrypto.base64decode(ah.substring(ah.indexOf(" ") + 1));
            String[] parts = decoded.split(":");
            String userName = parts[0];
            String password = parts[1];
            String hashedPassword = WeakCrypto.hashPassword(password);
            AppUser user = appUserRepo.findByName(userName).orElseThrow();
            if (!user.getHashedPassword().equals(hashedPassword)) {
                throw new Exception();
            }
            request.getSession().setAttribute(SESSION_USER_NAME, userName);
            return user;
        } catch (Exception e) {
            logOut(request);
            throw ClientErrors.unauthorized();
        }
    }
}
```

Simple role-based security. Return 403 if no permission

Retrieve header from request

Retrieve app user from database by name

Ignore „Basic“ prefix, base64 decode and split into parts

Compare password hash with stored hash

If auth successful, store user in session (see next slide)

Otherwise, return 401 response

Sessions allow to store *transient* data across requests.

- A session is initialized automatically upon the first request from a specific client.
- The generated session is identified by a *cookie*.
  → Clients must activate cookies to be able to make use of sessions.
- Sessions should only store *transient* data (e.g., the logged-in user) but are not a replacement for persistent storage (i.e., the database)!
- The current session may be accessed from an HttpServletRequest.
  - `getAttribute` and `setAttribute` are offered for reading/writing session data.

Constant for custom session attribute

```
@Service public class AuthService {
    private static final String SESSION_USER_NAME = "mensa-session-user-name";

    public AppUser getAuthenticatedUser(HttpServletRequest request) {
        Object sessionUserName = request.getSession().getAttribute(SESSION_USER_NAME);
    … }

    public AppUser logIn(HttpServletRequest request) { …
        request.getSession().setAttribute(SESSION_USER_NAME, userName);
    … }

    public void logOut(HttpServletRequest request) {
        request.getSession().setAttribute(SESSION_USER_NAME, null);
    }
}
```

Reading a session attribute

Setting a session attribute

Deleting a session attribute

App entrypoint and configuration

The entrypoint is a regular `main` method.

Marker annotation required for automatic component detection

```
@SpringBootApplication public class MensaBackend {
    public static void main(String[] args) {
        SpringApplication.run(MensaBackend.class, args);
    }
}
```

- `SpringApplication` dynamically finds components (e.g., controllers, services, repositories) in the current package and initializes them.

To allow arbitrary clients use the API, endpoints must accept *cross-origin requests*.

- By default, Spring Boot allows incoming REST requests only from the *same server*.
- Since we deploy frontend and backend on different servers, *cross-origin requests* must be activated via the *CORS (cross-origin resource sharing)* headers.
- More fine-grained security control is required for real productive applications.

```
@Configuration public class WebConfiguration implements WebMvcConfigurer {
    @Override public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("*");
    }
}
```

Wildcard access to all API endpoints

Adding API documentation requires minimal configuration.

- **Step 1**: Add dependency to pom.xml

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.8.4</version>
</dependency>
```

- **Step 2**: Add a configuration class

```
@Configuration @SecurityScheme(
        type = SecuritySchemeType.HTTP,
        name = "basicAuth",
        scheme = "basic")
public class SpringDocConfig { }
```

Here, we add a security scheme for Basic authentication

- **Step 3**: Add specific annotations to controllers

Swagger will not require authentication for listing schools

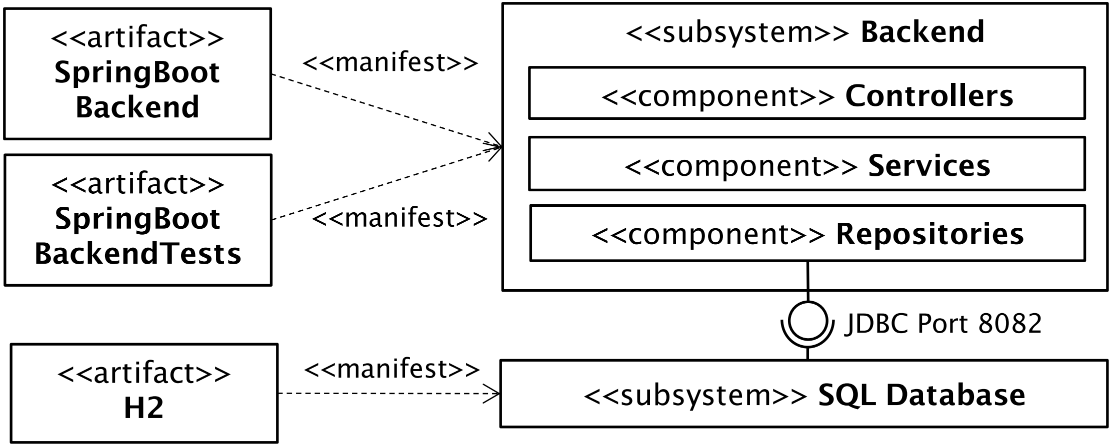
```
@RestController public class SchoolController {
    @GetMapping(value = "schools/{id}")
    public SchoolDto getSchool(@PathVariable("id") long id) { … }

    @SecurityRequirement(name = "basicAuth")
    @PostMapping(value = "schools")
    public SchoolDto createSchool(…) { … }
}
```

Swagger will require authentication for creating schools

02_01_mensa_backend/
pom.xml
…/SpringDocConfig.java
…/controllers/SchoolController.java

Automated tests are indispensable for the backend.

- Strictly following the *unit* test principle, we would have to test layers separately.
  - E.g., test the controllers mocking the service layers.
- The tests shown here are rather *integration* tests.
  - The three backend layers are tested together.
  - Temporary in-memory database *H2* is used instead of real persistent database.
- Architectural schema of test environment:



H2 is an SQL database that may run in the main memory.

- Following dependency needs to be added to `pom.xml` with *test scope*:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

- H2 is started automatically in the background of the test runner process.
- Default credentials: `sa / password`
- Specific JDBC URLs, e.g., `jdbc:h2:mem:public` (see next slide)

Application.properties:

```
spring.application.name=mensa_backend
spring.datasource.url=jdbc:mariadb://127.0.0.1:3306/mensa?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=asdf1234
spring.jpa.hibernate.ddl-auto=update
logging.level.org.springframework.web.filter.CommonsRequestLoggingFilter=debug
```

Application-test.properties:

```
spring.application.name=mensa_backend_tests
spring.datasource.url=jdbc:h2:mem:public
spring.datasource.username=sa
spring.datasource.password=password
```

- The properties mechanism supports profiles.
- The test profile defined here overrides *some* of the base properties (but keeps the rest).
- Test cases may individually select a profile through an annotation:

Activate test profile
→ Look up application-test.properties in classpath

```
@ActiveProfiles("test")
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
public abstract class MensaControllerTestBase { … }
```

The test base manages default users and their sessions.

```
public abstract class MensaControllerTestBase {
    @BeforeEach public void beforeEach() {
        createAppUser(ADMIN_USERNAME, ADMIN_PASSWORD, Roles.ADMIN, Roles.MANAGER, Roles.USER);
        createAppUser(MANAGER_USERNAME, MANAGER_PASSWORD, Roles.MANAGER, Roles.USER);
        createAppUser(USER_USERNAME, USER_PASSWORD, Roles.USER);
    … }

    private void createAppUser(String userName, String password, String... roles) {
        // create app user and store in repository
    }

    protected MockHttpServletRequest mockRequest(String username, String password) {
        MockHttpServletRequest request = new MockHttpServletRequest();
        // restore session of same user, if any
        return request;
    }

    protected HttpServletRequest admin0() {
        return mockRequest(ADMIN_USERNAME, ADMIN_PASSWORD);
    }
    protected HttpServletRequest manager0() {
        return mockRequest(MANAGER_USERNAME, MANAGER_PASSWORD);
    }
    protected HttpServletRequest user0() {
        return mockRequest(USER_USERNAME, USER_PASSWORD);
    }
}
```

Initially create one default user per role

Use SpringBoot‘s mock request class for simulating HTTP requests

In concrete tests, controller requests can be made on behalf of the default users.

Example test case

```java
public class DishControllerTest extends MensaControllerTestBase {
    @Autowired private SchoolController schoolController;
    @Autowired private DishController controller;
    private long schoolId;

    @BeforeEach @Override public void beforeEach() {
        super.beforeEach();
        schoolId = schoolController.createSchool(admin0(),
                                             new CreateSchoolDto("school1")).id();
    }

    @Test public void testListEmpty() {
        assertTrue(controller.listDishes(user0(), schoolId).isEmpty());
    }

    @Test public void testCreateUpdateGet() {
        long id1 = controller.createDish(manager0(), schoolId,
                                         new CreateDishDto("dish1", "nice")).id();
        DishDto updated = controller.updateDish(manager0(), schoolId, id1,
                                         new UpdateDishDto("dish1u", "delicious"));
        assertEquals("dish1u", updated.name());
        assertEquals("delicious", updated.description());
        DishDto dto = controller.getDish(user0(), schoolId, id1);
        assertEquals("dish1u", dto.name());
        assertEquals("delicious", dto.description());
    }
}
```

In addition to „unit under test“ controller, more controllers are required here

Work with default users defined in base class

Create initial data used by all test cases

Test initial behavior (a new school has 0 dishes)

Updated values have to be returned by both PUT and subsequent GET

Test coverage reveals spots not tested yet.

Run 'java in mensa-backend' with Coverage

Element	Class, %	Method, %	Line, %	Branch, %
oth.ics.wtp.mensa.backend	92% (35/38)	97% (134/138)	92% (288/310)	60% (30/50)
controllers	100% (6/6)	100% (31/31)	100% (68/68)	100% (0/0)
dtos	100% (12/12)	100% (12/12)	100% (12/12)	100% (0/0)
entities	100% (6/6)	100% (33/33)	100% (49/49)	100% (2/2)
repositories	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
services	100% (7/7)	100% (47/47)	90% (144/159)	56% (25/44)
AppUserService	100% (1/1)	100% (7/7)	90% (18/20)	50% (5/10)
AuthService	100% (1/1)	100% (6/6)	85% (24/28)	58% (7/12)
CommentService	100% (1/1)	100% (9/9)	96% (26/27)	50% (1/2)
DishService	100% (1/1)	100% (8/8)	82% (24/29)	50% (5/10)

```
    public DishDto update(long schoolId,
        if (!schoolRepo.existsById(schoo
            throw ClientErrors.schoolNot
        }
        Dish dish = dishRepo.findBySchoolIdAndId(schoolId, id)
                .orElseThrow(() -> ClientErrors.dishNotFound(id));
        dish.setName(updateDish.name());
        dish.setDescription(updateDish.description());
        return toDto(dishRepo.save(dish));
    }

    public void delete(long schoolId, long id) {
        if (!schoolRepo.existsById(schoolId)) {
            throw ClientErrors.schoolNotFound(schoolId);
        }
        if (!dishRepo.existsBySchoolIdAndId(schoolId, id)) {
            throw ClientErrors.dishNotFound(id);
        }
        dishRepo.deleteById(id);
    }
```

Error conditions should be addressed by additional test cases

- Main metric applied for your submission: Line coverage of overall project
- >80% is okay, >90% is perfect

New requirement: Update schools

- We create a new *feature branch* `feature/update-schools`
- We extend our API by a method to *update* schools.
    - The only relevant property so far is the school‘s *name*. (IDs shouldn‘t be updated)
- We implement the requirement in the following layers and representations:
    - Controller
    - DTO
    - Service
    - Entity (if required)
    - Repository (if required)
- We add a *JUnit test* for the method.
- We execute the backend locally and test the new method with *Swagger UI*.

HOCHSCHULE REGENSBURG UNIVERSITY OF APPLIED SCIENCES

REGENSBURG

Spring Boot is a powerful framework for implementing REST APIs.

- Convention over configuration
- Loose coupling by dependency injection
- SQL and REST are abstracted away → focus on Java classes (and annotations)
- We here considered a reference architecture with three layers
  - Repositories, services, controllers
  - Explicit distinction between entities and DTOs
- SpringDoc allows to generate Swagger/OpenAPI documentation (but we‘ve only scratched the surface)
- Testing the backend is indispensable. We discussed an integration test approach.
- Not considered in this project: Advanced authentication and security
  - Handcrafted Basic auth (Spring provides a, more difficult to use, integrated auth)
  - OAuth and JWT tokens
  - Fine-grained CORS filtering
  - HTTPS
  - Security tests
- You are recommended to implement the presented reference architecture also in your project! Re-using generic code parts (e.g., auth system, configuration) is allowed.

References

- [Hinkula 2022] Juha Hinkula: Full Stack Development with Spring Boot 3 and React, Packt, 2022
- [SpringBoot 2025] Spring Boot online documentation: https://docs.spring.io/spring-boot/index.html
- [ChatGPT1] ChatGPT (https://chatgpt.com/) with prompt: “I want to create a slide about history of Java web development frameworks and standards in my lecture about Spring Boot. Please give me a 8-bullet-point summary”
- [ChatGPT2] ChatGPT (https://chatgpt.com/) with prompt: “I want to create a slide about "Spring and Spring boot". Please summarize the most important characteristics and their relationships to other components (e.g., JAX, JPA, Tomcat)” [result shortened]