

Bachelor of Science (BSc) in Informatik
Modul Advanced Software Engineering 1 (ASE2)

LE 01 – Spring Framework

Spring Boot Einführung

Hello Rest

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

<https://www.zhaw.ch/de/engineering/institute-zentren/init/>

Agenda Spring Boot Einführung

1. LE01: Spring Boot – Hello Rest
 1. Einführung und Ziele
 2. Erstellen eines Rest-Projekts
2. LE02: Datenbank Migration
 1. Spring Data mit Flyway
 2. Spring Data mit Liquibase
3. LE03-1 Spring Profiles
 1. Auto Konfiguration
 2. Properties
 3. Spring Profiles
4. LE03-2 Deployment mit Docker
 1. Build jar und Docker Container
 2. Docker compose
5. LE04-1: Einführung Spring MVC
 1. Controller erstellen
 2. @RequestMapping, Params, etc
 3. RestController erstellen
6. LE04-1: Spring Security
 1. Architektur
 2. Authentisierung
 3. Autorisierung
 4. Stateless Backend mit JWT

Lernziele LE 01-04 – Spring Framework

- Der Studierenden
 - können mittels Initializr ein Spring Boot Projekt erstellen
 - können die automatisch generierte REST Schnittstelle anwenden
 - können mittels Flyway und Liquibase Datenbank Refaktorisierung durchführen
 - Wissen wie die Spring Boot Autokonfiguration funktioniert
 - Können Profile einrichten und anwenden
 - Können Spring MVC für eigenen Controller bzw. RestController inkl. Exceptionhandling
 - Können Spring Security aktivieren und einsetzen

Spring Boot Quellen

- **Spring IO**
 - Spring IO Website: <http://www.spring.io>
 - Spring Projects: <http://spring.io/projects>
 - Spring Guides: <http://spring.io/guides>
 - Spring Document: <http://spring.io/guides>
 - Spring Boot Docs: <http://docs.spring.io/spring-boot/docs/current/ref...>
 - Spring Boot API: <http://docs.spring.io/spring-boot/docs/current/api...>
- **Stay Connected**
 - Twitter: twitter.com/springcentral
 - YouTube: spring.io/video
 - Questions: spring.io/questions
 - JIRA: jira.spring.io
 - Blog: <http://spring.io/blog>

Agenda LE01: Spring Boot – Hello Rest

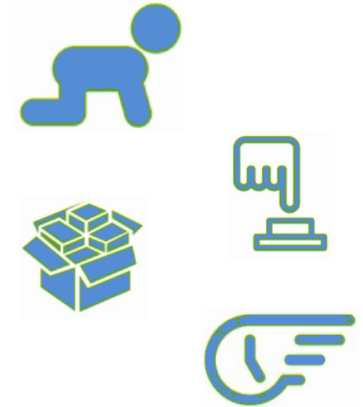
- Spring Boot Einführung
- Hands-on 0: Erstellen eines Projekts mittels Spring Initializr
- Hands-on 1: Einfachster Controller mit UnitTest
- Hands-on 2: Automatische REST-Schnittstelle

Was ist Spring Boot (1)

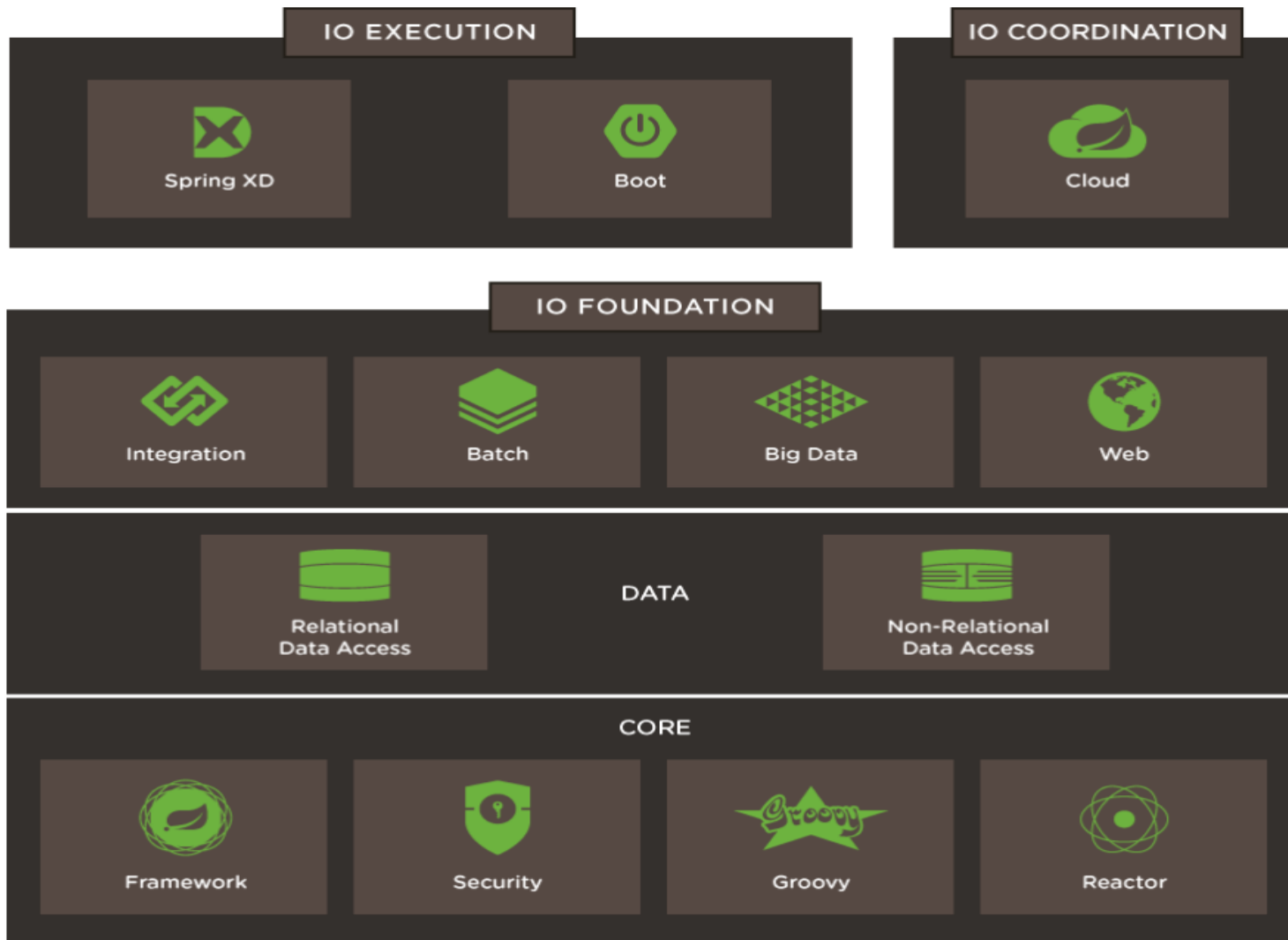
- Mittlerweile der Standard für Spring Projekte
- Ausgangslage
 - Komplexe XML-Konfiguration
 - Abhängigkeitsverwaltung (grosse Anzahl von Bibliotheken)
 - Applikationsserver (zusätzliche Komplexität, viele Produkte)
- Veränderungen in der IT
 - DevOps
 - Agile Entwicklung
 - Container
 - Cloud

Was ist Spring Boot (2)

- Ziele
 - Einfacher Einstieg
 - Convention over Configuration
 - Vollständige Applikationspakete
 - Schnelle Entwicklungszyklen



Was ist Spring Boot (3)



Was ist Spring Boot (4) - Beispiel

- Vollständige HelloWorld Anwendung
- Kann mittels Run as -> Java Application gestartet werden

```
// same as @Configuration @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
@Controller
public class DemoApplication {

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

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World";
    }
}
```

Einfacher Einstieg

- Spring Initializr
 - Auswahl gewünschter Pakete
 - Fertige Build-Konfiguration mit Beispielklassen
 - Eigene Bedürfnisse: Java/Groovy/Kotlin – Gradle/Maven
- Groovy Skript
 - Start von einfachem Skript über das Spring CLI

Convention over Configuration

- Automatische Konfiguration
 - Starterpakete mit automatischer Konfiguration
<http://docs.spring.io/spring-boot/docs/current/reference/html/auto-configuration-classes.html>
 - Erkennung von Bibliotheken im Classpath (H2 / Memory Databank)
 - Pakete
 - Web/REST
 - JPA/Persistenz
 - JMS
 - Cloud

Applikationspakete

- Vollständige Applikationspakete
 - Einzelne WAR/JAR
 - Deployment als Standalone-Service
 - Auch Deployment auf Applikationsserver
 - Alle Abhängigkeiten enthalten



Konfigurationsmöglichkeiten

- Property-Dateien
- Umgebungsvariablen (Container)
- XML/YAML/JSON/...



Entwicklungszyklus

- Deployment
 - Hot-Reload (spring-boot-devtools)
 - Hot Swapping
 - JRebel: <https://zeroturnaround.com/software/jrebel/>
 - Spring-Loaded: <https://github.com/spring-projects/spring-loaded>
 - Keine Konfiguration via XML
 - Code-driven



Erstellen eines Projekts

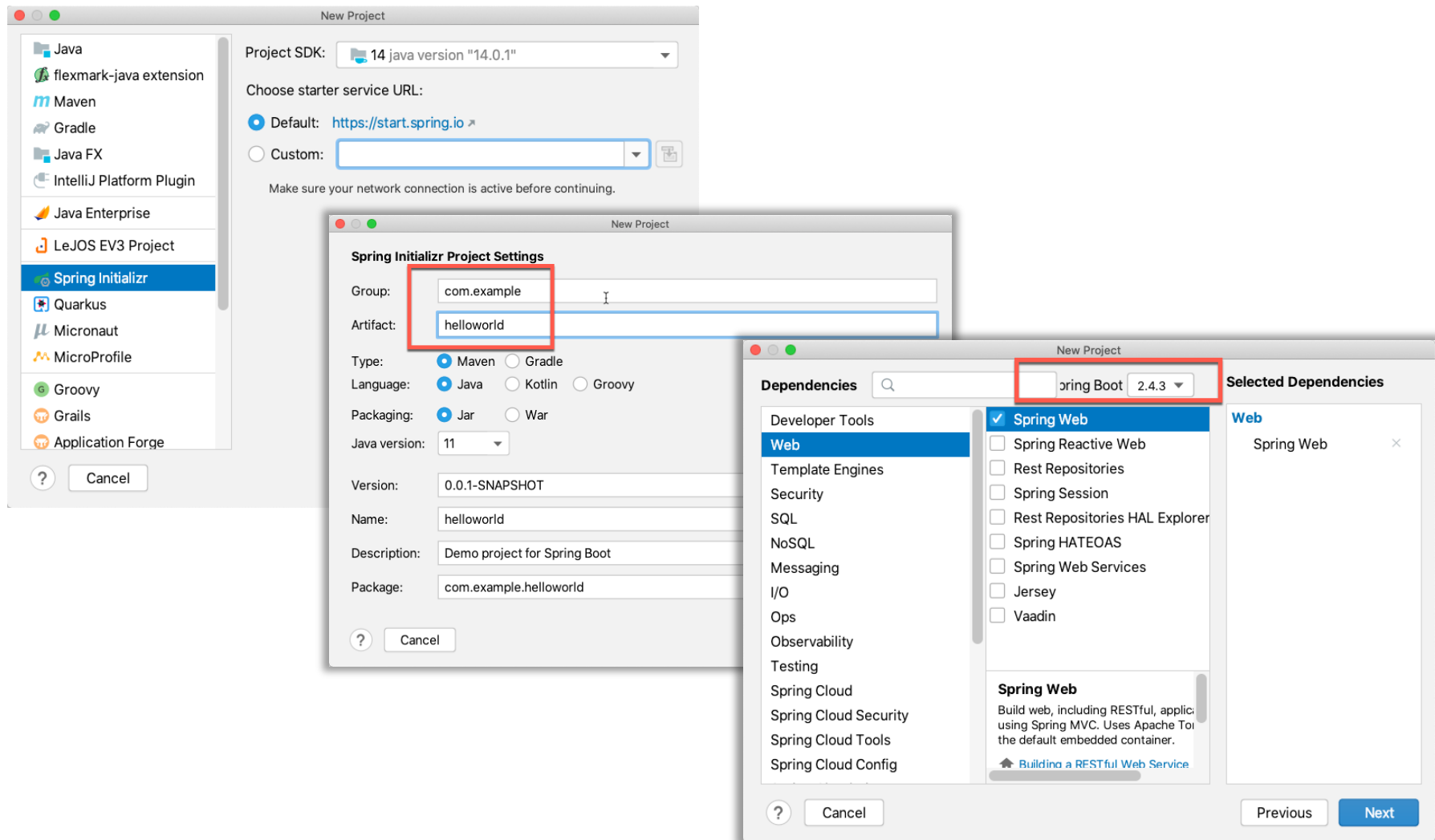
- Direkt aus IDE mittels Spring->Starter
- Mittels Web Seite Spring Initializr -> `start.spring.io`
- Aufbau der Beispiel-Projekte ASE2 Spring Boot
 - Maven Projekt mit verschiedenen Branches

<https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest>

Hands-on 0

- Erstellen eines Projekts mittels Spring Initializr

Hands-on --- File new Project ...



Was wurde erstellt?

The screenshot shows an IDE window for a project named "hello-world". The main editor displays the `HelloWorldApplication.java` file, which contains the following code:

```
1 package com.example.helloworld;  
2  
3 import ...  
4  
5 @SpringBootApplication  
6 public class HelloWorldApplication {  
7  
8     public static void main(String[] args) {  
9         SpringApplication.run(HelloWorldApplication.class, args);  
10    }  
11 }
```

Annotations with red arrows point to various files in the project structure:

- main**: Points to the `main` method in `HelloWorldApplication.java`.
- mvc web resources**: Points to the `static` directory in the `resources` folder.
- mvc jsp or thymeleaf templates**: Points to the `templates` directory in the `resources` folder.
- Parametrierung der App**: Points to the `application.properties` file in the `resources` folder.
- Dummy Unit Test für den Context**: Points to the `HelloWorldApplicationTest` file in the `test` folder.
- Vorbereitetes .gitignore**: Points to the `.gitignore` file in the root of the project.
- Maven wrapper**: Points to the `mvnw` file in the root of the project.
- Maven dependencies und plugins**: Points to the `pom.xml` file in the root of the project.

The Maven sidebar on the right shows the project structure, including the `hello-world` project, its lifecycle, plugins, and dependencies.

SpringApplication

- Die statische run Methode von *SpringApplication* ermöglicht den Start der *SpringBoot* Anwendung
 - Der *Application Context* wird erstellt und alle Singleton-Beans werden geladen
 - Registriert eine *CommandLinePropertySource* Bean
 - Triggert eine *CommandLineRunner* Beans
- Die Annotation *@SpringBootApplication* beinhaltet folgende Annotationen
 - *@Configuration*
 - *@ComponentScan*
Spring sucht nach Annotationen wie Configuration, Controller, Service, Repository, RestController
 - *@EnableAutoConfiguration*
Veranlasst SpringBoot den Anwendungskontext automatisch zu konfigurieren
Die Konfiguration basiert auf den eingebundenen Dependencies (z.B. H2 DB)

Der CommandLineRunner

- Falls ein Code **vor** der *run-Methode* von *SpringApplications* ablaufen soll, dann kann dies wie folgt erreicht werden:

```
@Component
public class MyBean implements CommandLineRunner {
    public void run(String[] args) {
        ...
    }
}
```


Analyse des generierten pom's








- Verweis auf übergeordnetes Starter-pom

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.3</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

- Plugin für run und package

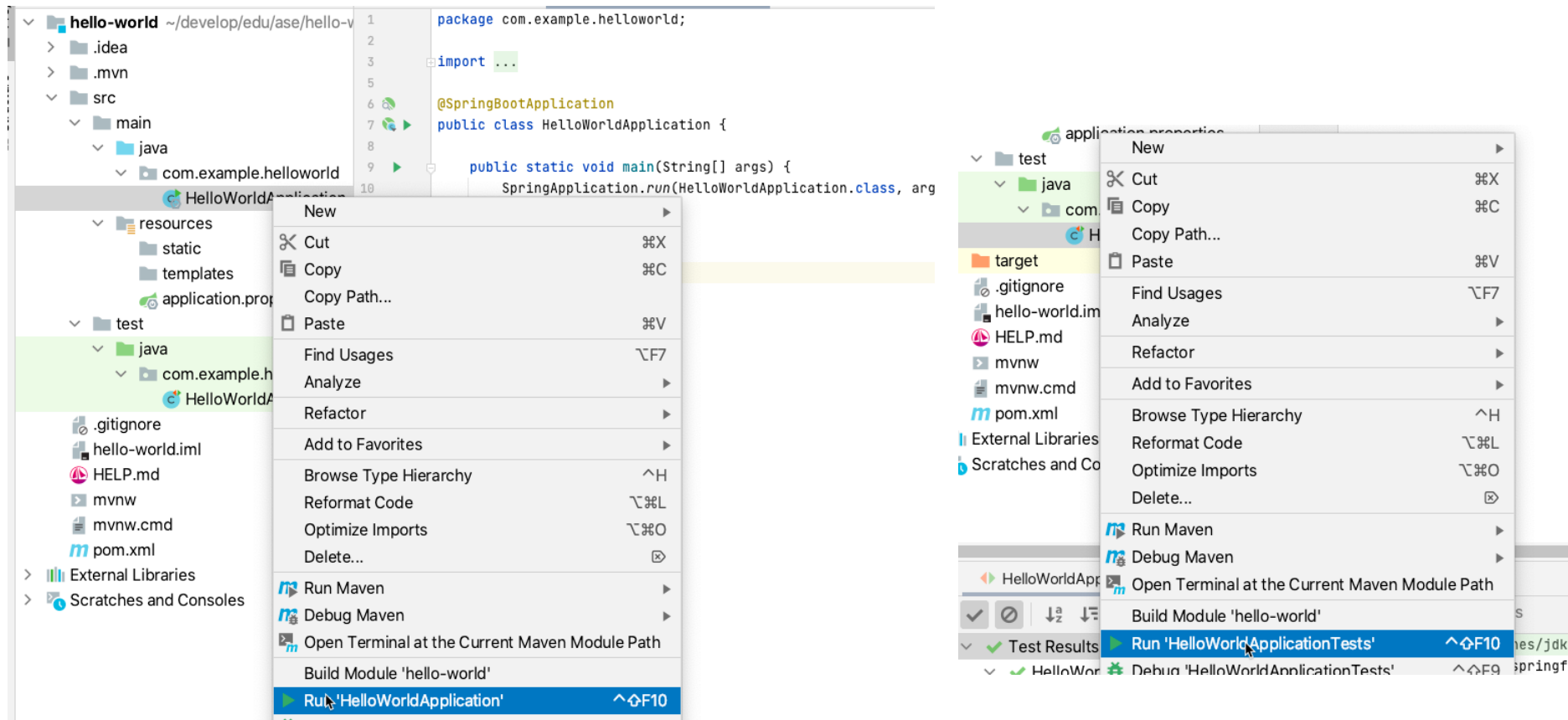
```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

▼  spring-boot (org.springframework.boot)

-  spring-boot:build-image
-  spring-boot:build-info
-  spring-boot:help
-  spring-boot:repackage
-  **spring-boot:run**
-  spring-boot:start
-  spring-boot:stop

Test der generierten Anwendung

- App und Test starten mittels Run as oder Debug as



Warum benötigen wir keinen Tomcat?

- Spring Boot hat einen integrierten Web Server
<https://docs.spring.io/spring-boot/docs/2.1.9.RELEASE/reference/html/howto-embedded-web-servers.html>
 - Dieser wird *automatisch* gestartet
- Der *Default-Web-Server* für *Spring-MVC* ist *Tomcat*
 - Jetty oder
 - Undertow sind auch möglich
- Der *Default-Web-Server* für Spring-Reactive (ab Spring 5 bzw. Spring Boot 2 mit Webflux) ist Netty
 - Tomcat, Jetty oder Undertow sind auch möglich

Anwendung von Maven

- Spring Boot wurde für eine bestimmte JDK Version generiert (JDK8, JDK11, JDK14/15)

```
<properties>
|   <java.version>11</java.version>
</properties>
```

 - Einstellen der Entwicklungsumgebung für dieses JDK (z.B. IntelliJ -> Project Structure)
 - Einstellen der Konsole (Terminal) für dieses JDK (**java -version** muss dem eingestellten JDK im pom entsprechen)
 - Ggf. JDK Konsolen-Switch einrichten für die dynamische Umschaltung des JDK's
- Starten mittels mvn spring-boot:run

```
501 mbach:hello-world $ mvn spring-boot:run
```


Default Goal

- Default Goal in pom einrichten

```
<build>  
  <defaultGoal>spring-boot:run</defaultGoal>  
  <plugins>
```

- Anschliessend kann mit mvn gestartet werden

Hands-on 1

- Einfachster Controller mit UnitTest

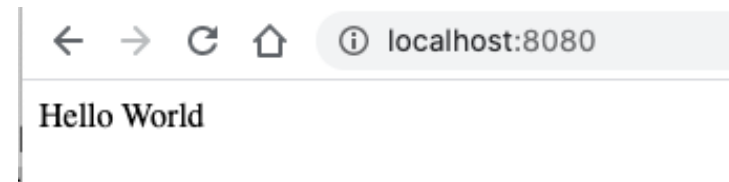
Einen Controller einrichten

```
@SpringBootApplication
@Controller
public class HelloWorldApplication {

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

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World";
    }
}
```

- Im Browser: <http://localhost:8080> eintippen



Einen Unit Test erstellen

```
@SpringBootTest
@AutoConfigureMockMvc
public class HelloControllerTest {

    @Autowired
    private MockMvc mvc;

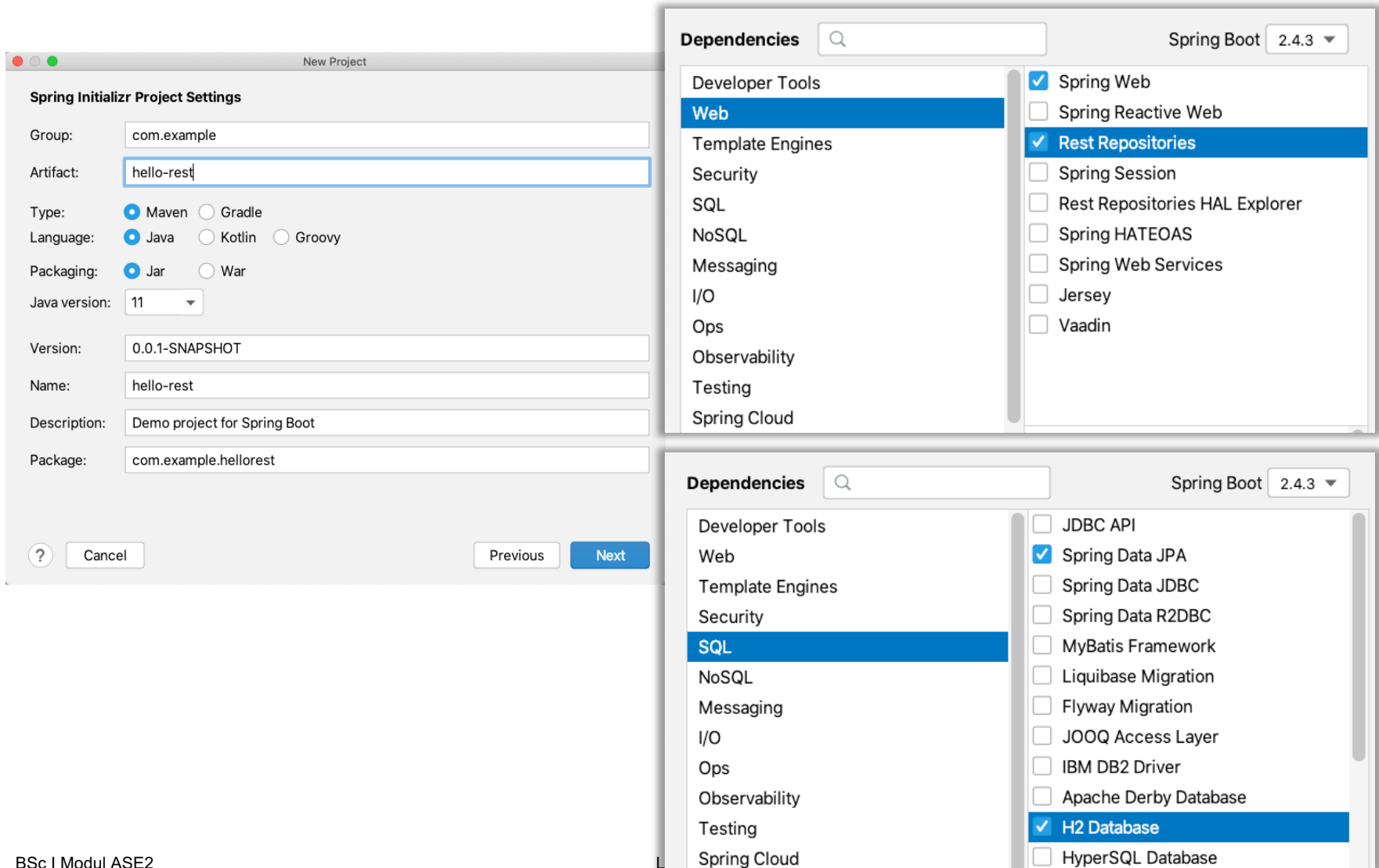
    @Test
    public void getHello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/")
            .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string(equalTo("Hello World")));
    }
}
```

✓	Test Results	391 ms
✓	HelloControllerTest	391 ms
✓	getHello()	391 ms

Hands-on 2

- Automatische REST-Schnittstelle
 - Datenbankzugriff
 - H2 Datenbank
 - H2 Konsole
 - OpenApi 3.0
 - Postman

Neues Projekt mit Initializr



The image displays two screenshots of the Spring Initializr web interface. The left screenshot shows the 'New Project' settings form, and the right screenshot shows the 'Dependencies' selection screen.

Spring Initializr Project Settings

Group:

Artifact:

Type: ☒ Maven ☐ Gradle

Language: ☒ Java ☐ Kotlin ☐ Groovy

Packaging: ☒ Jar ☐ War

Java version:

Version:

Name:

Description:

Package:

? Cancel Previous Next

Dependencies Spring Boot 2.4.3

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Ops

Observability

Testing

Spring Cloud

☒ Spring Web

☐ Spring Reactive Web

☒ Rest Repositories

☐ Spring Session

☐ Rest Repositories HAL Explorer

☐ Spring HATEOAS

☐ Spring Web Services

☐ Jersey

☐ Vaadin

Dependencies Spring Boot 2.4.3

Developer Tools

Web

Template Engines

Security

SQL

NoSQL

Messaging

I/O

Ops

Observability

Testing

Spring Cloud

☐ JDBC API

☒ Spring Data JPA

☐ Spring Data JDBC

☐ Spring Data R2DBC

☐ MyBatis Framework

☐ Liquibase Migration

☐ Flyway Migration

☐ JOOQ Access Layer

☐ IBM DB2 Driver

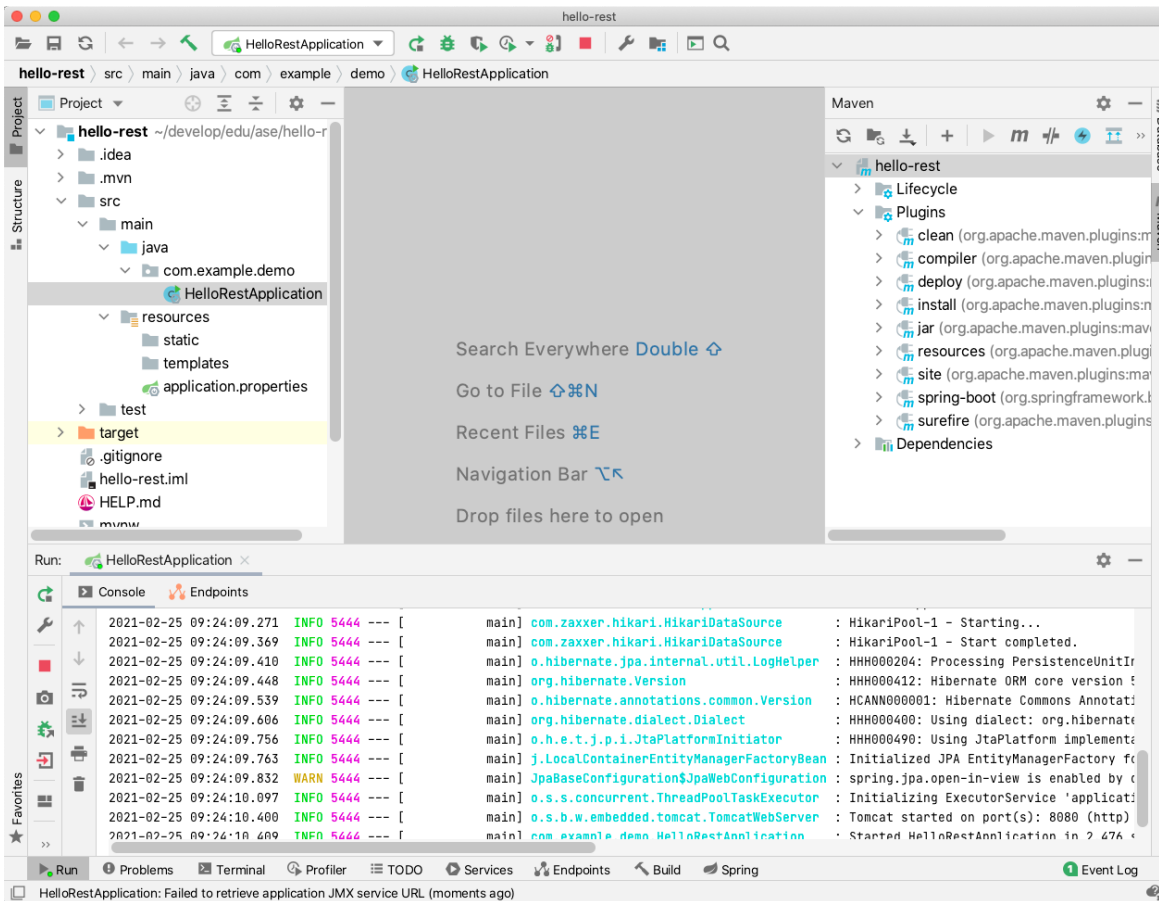
☐ Apache Derby Database

☒ H2 Database

☐ HyperSQL Database

Generiertes Projekt

- Starten



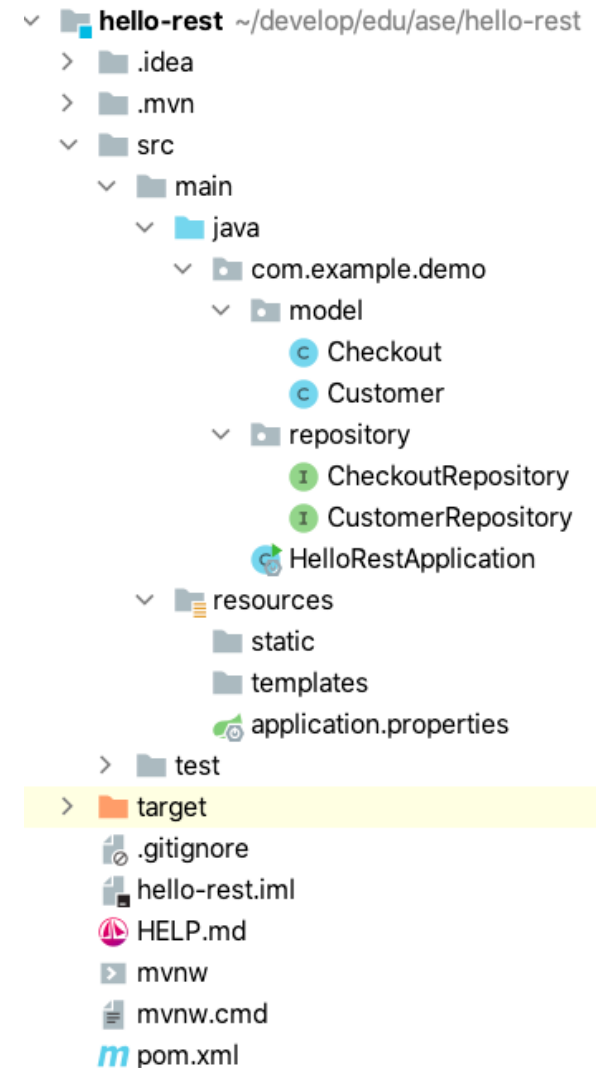
Pom erweitern mit OpenApi 3.0

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.5.4</version>
</dependency>

<!-- Open API for Automatic Rest Interfaces -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-data-rest</artifactId>
  <version>1.5.4</version>
</dependency>
```


Projektstruktur

- Model Klassen in Model Package
- Repository Klassen in Repository Package



Datenzugriff

- Konzept
- Standard SQL-Datenbank: H2 (in-Memory)
- Programmierung via JPA (Java Persistence API)
- Basiert auf Hibernate
- Austausch gegen andere SQL-Datenbanken: MySQL, Postgres, ...
- Automatisches Generieren von Webschnittstellen (REST)

Java Persistence API (JPA)

- Modellklassen
 - Definition von Eigenschaften
 - Definitionen von Beziehungen
 - Objektorientierung
- Repository
 - Zugriff auf Instanzen der Modellklasse
 - Automatisches Generieren von Webschnittstellen (REST)
 - Die Jar-Datei **spring-boot-starter-data-rest** stellt automatisch eine Restschnittstelle basierend auf den Repositories zur Verfügung
<https://docs.spring.io/spring-data/rest/docs/current/reference/html/> - customizing-sdr
 - Kann mittels Annotationen angepasst werden
 - Mittels **Swagger** können die Schnittstellen automatisiert dokumentiert werden

Application Klasse

```
@SpringBootApplication
public class HelloRestApplication {

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

    @PostConstruct
    public void afterInit() {
        System.out.println("\n\nEnter in Browser:\nhttp://localhost:8080 \n" +
            "http://localhost:8080/v3/api-docs\n" +
            "http://localhost:8080/swagger-ui.html \n" +
            "http://localhost:8080/h2-console " + "'" +
            "-> mit Generic H2 (Embedded), org.h2.Driver, jdbc:h2:mem:testdb und sa \n\n");
    }
}
```

Model-Klasse Customer und Checkout

```
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;

    private String firstname;
    private String lastname;

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname)
    {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

```
@Entity
public class Checkout {

    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Customer customer;

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

Repository Klassen

- Zugriff via `org.springframework.data.repository.CrudRepository`

```
/**
 * Repository class for Customer
 */
public interface CustomerRepository extends
    CrudRepository<Customer, Long> {
}
```

```
/**
 * Repository class for Checkout
 */
public interface CheckoutRepository extends
    CrudRepository<Checkout, Long> {
}
```

```
CrudRepository<T, ID extends Serializable>
● A save(S) <S extends T> : S
● A save(Iterable<S>) <S extends T> : Iterable<S>
● A findOne(ID) : T
● A exists(ID) : boolean
● A findAll() : Iterable<T>
● A findAll(Iterable<ID>) : Iterable<T>
● A count() : long
● A delete(ID) : void
● A delete(T) : void
● A delete(Iterable<? extends T>) : void
● A deleteAll() : void
```

application.properties

```
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true

springdoc.swagger-ui.path=/swagger-ui.html
```

Installation Postman

- <https://www.postman.com/downloads/>



Get Request auf den Root Endpoint

The screenshot shows a REST client interface with the following components:

- Method:** GET (selected from a dropdown)
- URL:** http://localhost:8080
- Params:** (empty)
- Buttons:** Send (highlighted), Save
- Tabs:** Authorization, Headers, Body (selected), Pre-request Script, Tests
- Code:** (button)
- Type:** No Auth (selected from a dropdown)
- Status:** 200 OK
- Time:** 185 ms
- Body:** (selected from a dropdown)
- Format:** JSON (selected from a dropdown)
- View:** Pretty (selected from a dropdown)
- Raw:** (button)
- Preview:** (button)
- JSON:** (button)
- Copy:** (button)
- Search:** (button)

The response body is shown in JSON format:

```
1 {  
2   "_links": {  
3     "customers": {  
4       "href": "http://localhost:8080/customers"  
5     },  
6     "checkouts": {  
7       "href": "http://localhost:8080/checkouts"  
8     },  
9     "profile": {  
10      "href": "http://localhost:8080/profile"  
11    }  
12  }  
13 }
```

Daten einfügen mittels Post Request

The screenshot shows a REST client interface with the following elements:

- Method:** POST (selected from a dropdown)
- URL:** http://localhost:8080/customers
- Params:** (empty)
- Send:** (button)
- Body Tab:** Selected, showing a JSON body:

```
{  
  "firstname": "Matthias",  
  "lastname": "Bachmann"  
}
```
- Form Data:** (radio buttons for form-data, x-www-form-urlencoded, raw, binary)
- Content Type:** JSON (application/json) (selected from a dropdown)

- Response von Spring

The screenshot shows the response of a POST request in a REST client interface. The response is displayed in the 'Body' tab, which is selected. The response is a JSON object with the following structure:

```
{  
  "firstname": "Matthias",  
  "lastname": "Bachmann",  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/customers/3"  
    },  
    "customer": {  
      "href": "http://localhost:8080/customers/3"  
    }  
  }  
}
```

Get-Request

http://localhost:8080/customers

```
{
  "_embedded": {
    "customers": [
      {
        "firstname": "Matthias",
        "lastname": "Bachmann",
        "_links": {
          "self": {
            "href": "http://localhost:8080/customers/1"
          },
          "customer": {
            "href": "http://localhost:8080/customers/1"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/customers"
    },
    "profile": {
      "href": "http://localhost:8080/profile/customers"
    }
  }
}
```

Daten einfügen mittels Post Request

POST Params

Authorization Headers (1) **Body** Pre-request Script Tests Code

☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary

```
1 {  
2   "customer": "http://localhost:8080/customers/1"  
3 }
```

Body Cookies Headers (4) Tests

Pretty Raw Preview JSON

```
1 {  
2   "_links": {  
3     "self": {  
4       "href": "http://localhost:8080/checkouts/1"  
5     },  
6     "checkout": {  
7       "href": "http://localhost:8080/checkouts/1"  
8     },  
9     "customer": {  
10      "href": "http://localhost:8080/checkouts/1/customer"  
11    }  
12  }  
13 }
```

Test H2 Database

- Eingabe von localhost/h2-console
- JDBC URL: jdbc:h2:mem:testdb

English ▾ [Preferences](#) [Tools](#) [Help](#)

Login

Saved Settings: Generic H2 (Embedded) ▾

Setting Name: Generic H2 (Embedded)

Save

Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

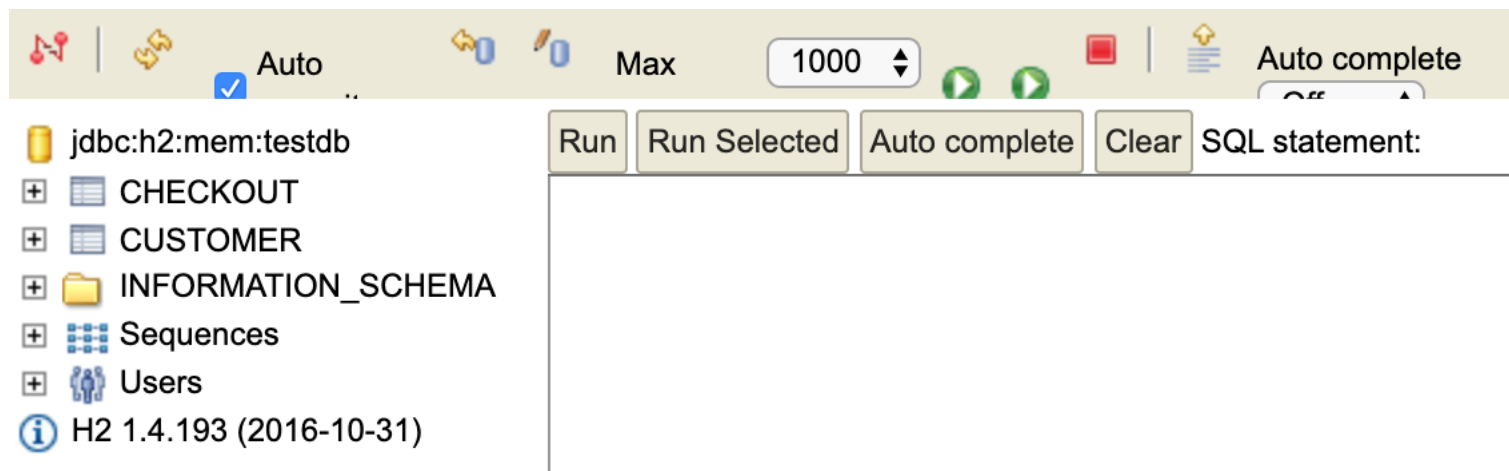
Password:

Connect

Test Connection

Test H2 Database

- Einsehen der Tabellen und Spalten
- Datenabfrage



Swagger

- <http://localhost:8080/swagger-ui.html>

The screenshot shows the Swagger UI interface. At the top, there is a green header bar with the Swagger logo (a green circle with a white curly brace) and the word "swagger" in white. To the right of the logo, there is a dropdown menu labeled "Select a spec" with "default" selected. Below the header, the main content area has a light gray background. It features a large heading "11 Spring Boot Rest" with a small "2.0" version indicator in a gray circle. Below the heading, there is a text block containing the base URL "[Base URL: localhost:8080/]" and a link "http://localhost:8080/v2/api-docs". Underneath, there is a section labeled "description" with three links: "name - Website", "Send email to name", and "Apache License Version 2.0". At the bottom, there is a list of API endpoints, each with a title, a description, and a right-pointing chevron icon. The endpoints are: "Checkout Entity" (Simple Jpa Repository), "Customer Entity" (Simple Jpa Repository), and "basic-error-controller" (Basic Error Controller).

swagger Select a spec default

11 Spring Boot Rest ^{2.0}

[Base URL: localhost:8080/]
<http://localhost:8080/v2/api-docs>

description

[name - Website](#)
[Send email to name](#)
[Apache License Version 2.0](#)

Checkout Entity Simple Jpa Repository >

Customer Entity Simple Jpa Repository >

basic-error-controller Basic Error Controller >

OpenApi

- <http://localhost:8080/v3/api-docs>

```
{
  swagger: "2.0",
  - info: {
    description: "description",
    version: "2.0",
    title: "11 Spring Boot Rest",
    - contact: {
      name: "name",
      url: "url",
      email: "email"
    },
    - license: {
      name: "Apache License Version 2.0",
      url: "https://github.com/springfox/springfox/blob/master/LICENSE"
    }
  },
  host: "localhost:8080",
  basePath: "/",
  - tags: [
    - {
      name: "Checkout Entity",
      description: "Simple Jpa Repository"
    },
    - {
      name: "Customer Entity",
      description: "Simple Jpa Repository"
    }
  ],
}
```


Unit Test (1) - Voraussetzung

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

```
@SpringBootTest
@AutoConfigureMockMvc
public abstract class AbstractTest {
    protected MockMvc mvc;
    @Autowired
    WebApplicationContext webApplicationContext;

    protected void setUp() {
        mvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }
}
```

<https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest/blob/master/src/test/java/com/example/demo/AbstractTest.java>

Unit Test (2)

- Aufruf einer REST Schnittstelle mittels `mvc.perform`

```
@Test
public void getOneCustomer() throws Exception {
    String uri = "/customers/1";
    MvcResult mvcResult = mvc.perform(MockMvcRequestBuilders.get(uri)
        .accept(MediaType.APPLICATION_JSON_VALUE, "application/hal+json"))
        .andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals(200, status);
}
```

<https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest/blob/master/src/test/java/com/example/demo/CustomerRestControllerTest.java>

Zusammenfassung hands-on 2

- H2 Datenbank
 - Initialisierung der Datenbank
- JPA Konfiguration
 - Anlegen von Modellklassen
 - Anlegen eines Repositories
 - Automatisches Erzeugen des Datenbankschemas
 - Automatische REST-Endpunkte
 - Abfrage mittels Postman
- Unit Test