

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

LE 03 – Spring Framework

Spring Boot

Profiles, Docker, Rest

Institut für Angewandte Informationstechnologie (InIT)

Walter Eich (eicw) / Matthias Bachmann (bacn)

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

Agenda

- LE03-1: Konfiguration Magic: Autokonfiguration, Properties, Profiles
- LE03-2: Spring mit Docker und Docker Compose

Konfiguration von Spring Boot

- Auto-Konfiguration
- Properties
- Profiles

Lernziele LE 03-1 – Konfigurationen

- Die Studierenden
 - Verstehen wie die Auto-Konfiguration in Spring-Boot funktioniert
 - Verstehen wie die Auto-Konfiguration überschrieben werden kann
 - Verstehen wie mit Hilfe von Properties eine Konfiguration beeinflusst werden kann
 - Können mittels Profiles unterschiedliche Konfigurationen erstellen

Auto Konfiguration

@EnableAutoConfiguration

```
@SpringBootApplication  
public class DemoApplication {}
```



```
@Configuration  
@ComponentScan  
@EnableAutoConfiguration  
public class DemoApplication {}
```

- Versucht die Spring-Boot Anwendung automatisch zu konfigurieren
- Falls eigene Beans zur Konfiguration einer bestimmten Komponente vorhanden sind, führt Spring-Boot keine Auto-Konfiguration durch
- Konfiguration durch normale @Configuration Klassen
- Normalerweise mit @ConditionalOnClass und
- @ConditionalOnMissingBean
- oder @ConditionalOnProperty

Auto Konfiguration

Registrierung falls in Classpath vorhanden

Starter Web

Spring MVC ist im
classpath

Dispatcher Servlet
wird registriert

H2 Datenbank
Dependency

H2 ist im classpath

Embedded
DataSource wird
registriert

Application.properties

Falls
spring.datasource.url
definiert

DataSource wird
registriert

Auto Konfiguration

Conditional Konfiguration

- **@ConditionalOnBean**
 - Die spezifizierte Bean wurde konfiguriert
- **@ConditionalOnMissingBean**
 - Die spezifizierte Bean wurde nicht konfiguriert
- **@ConditionalOnClass**
 - Die spezifizierte Klasse ist im Klassenpfad verfügbar
- **@ConditionalOnMissingClass**
 - Die spezifizierte Klasse ist im Klassenpfad nicht verfügbar

```
@Bean
@ConditionalOnMissingBean(JdbcOperations.class)
public JdbcTemplate jdbcTemplate () {
    return new JdbcTemplate(this.dataSource);
}
```

Auto Konfiguration

Anzeige der Auto Konfiguration

- Anzeige der Auto Konfiguration durch das **--debug** Flag

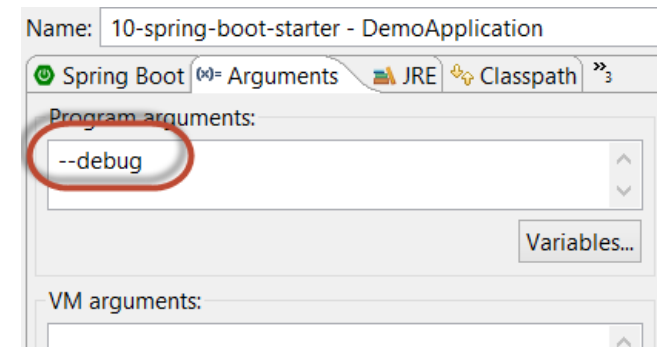
```
=====
AUTO-CONFIGURATION REPORT
=====
```

Positive matches:

```
-----

DispatcherServletAutoConfiguration matched:
- @ConditionalOnClass found required class
'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find
unwanted class (OnClassCondition)
- @ConditionalOnWebApplication (required) found StandardServletEnvironment
(OnWebApplicationCondition)
...
Negative matches:
-----

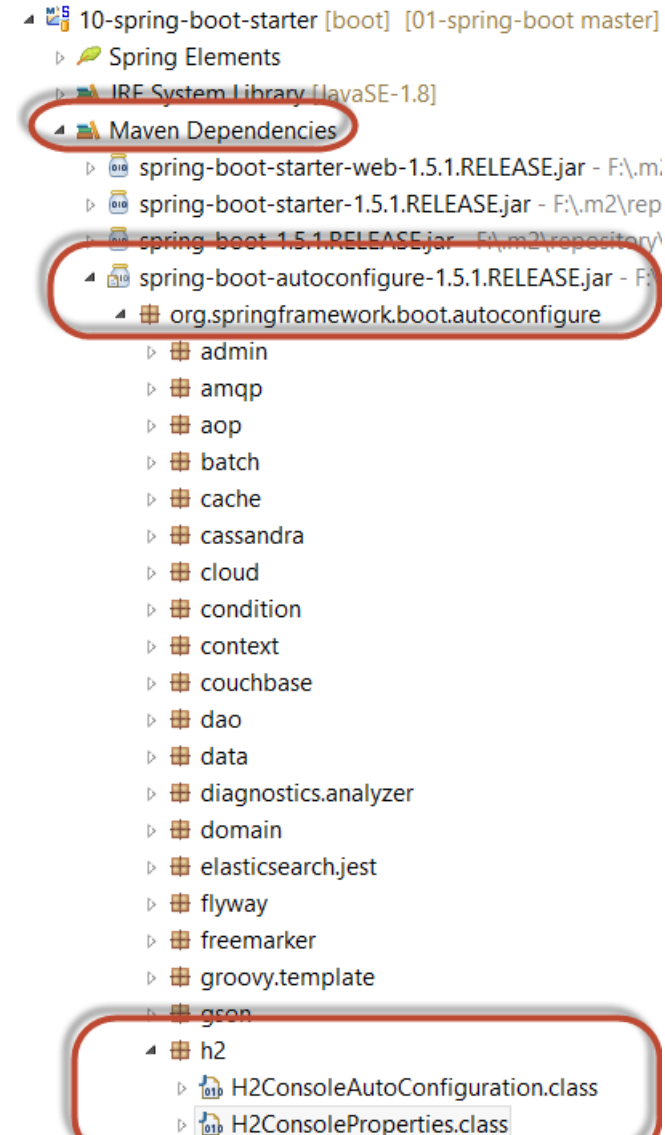
ActiveMQAutoConfiguration:
Did not match:
- @ConditionalOnClass did not find required classes 'javax.jms.ConnectionFactory',
'org.apache.activemq.ActiveMQConnectionFactory' (OnClassCondition)
```



Auto Konfiguration

spring-boot-autoconfigure

- Die **Auto-Konfiguration** ist gesteuert über Klassen im Klassenpfad oder über verfügbare Beans.
- Die Auto-Konfiguration kann überschrieben werden
 - durch Properties definiert in den Properties Klassen (**H2ConsoleProperties**)
- Die Auto Konfiguration kann durch eine eigene Konfigurationsklasse ersetzt werden.
- Die Auto-Konfiguration einzelner Klassen kann **ausgeschaltet** werden durch



Auto Konfiguration

Beispiel H2ConsoleProperties

- Spring Klasse: H2ConsoleProperties

```
public class H2ConsoleProperties {  
    private String path = "/h2-console";  
    private boolean enabled = false;  
    private final H2ConsoleProperties.Settings settings = new H2ConsoleProperties.Settings();  
  
    ...  
    public H2ConsoleProperties.Settings getSettings() { return this.settings; }  
  
    public static class Settings {  
        private boolean trace = false;  
        private boolean webAllowOthers = false;  
        private String webAdminPassword;  
    }  
}
```

- In application.properties

```
spring.h2.console.enabled=true  
spring.h2.console.settings.web-allow-others=true
```

Auto Konfiguration

Auto Konfiguration ausschalten

- Via Annotation

```
@Configuration
@EnableAutoConfiguration(excludeName="...", exclude={DataSourceAutoConfiguration.class})
public class AppConfiguration {
}
```

- Or via property

```
spring.autoconfigure.exclude =
...boot.autoconfigure.jdbc.DataSourceAutoConfiguration
```

Auto Konfiguration

Manuelles erstellen von Konfigurationen

- Mittels `@ContextConfiguration` können bestimmte Konfigurationsklassen für einen Test definiert werden

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = { DBConfig.class, MVConfig.class, JPAConfig.class })
class ConfigUnitTest {

    @Autowired
    ApplicationContext context;

}
```

- Mittels `@Import` können Konfigurationen gruppiert werden

```
@Configuration
@Import({DBConfig.class, MVConfig.class })
class TestConfiguration {

}
```

Properties

Externe Konfiguration der Komponenten

- Jeder Anwendungskontext hat eine Umgebung
- Abstraktion mittels Key/Value aus verschiedenen Quellen
- Properties
 - Erlauben die Konfiguration des Frameworks ohne Programmierung
 - Spring stellt für das Framework eine grosse Anzahl an Konfigurationsmöglichkeiten zur Verfügung
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>
 - Es können eigene Properties definiert werden

1. Core Properties

2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties
15. Devtools Properties
16. Testing Properties

Properties

Definition von externen Properties

- Die Properties einer Spring Application können via Command Line Argumente definiert werden:

```
java -jar target/*.jar --server.port=9000
```

- Oder durch eine externe Configuration (.properties oder .yaml)
 - application.properties

```
server.port=9000
```

- application.yml

```
server.port: 9000
```

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

Properties

Beispiel application.properties

- Beispiel einer application.properties Datei

```
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:file:./database;FILE_LOCK=FS
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
```

Properties

Konfiguration von Beans

- Konfiguration von Properties mit `@ConfigurationProperties`
- Default Values mit `@Value`: `127.0.0.1` ist Default
ggf. überschreiben in Datei `application.properties`
- Mittels `@Configuration` die `@ConfigurationProperties` aktivieren

```
@Component
@ConfigurationProperties(prefix="spring.datasource")
public class DataSourceProperties {

    @Value("${myOther.url:127.0.0.1}")
    private String myOtherUrl;

    public String getMyOtherUrl() {
        return myOtherUrl;
    }
}
```


Properties

Zugriff auf Properties mit Environment

- Mittels Environment env kann der Zugriff auf Profiles und Properties in Java durchgeführt werden.

```
@SpringBootApplication
public class HelloRestApplication {

    @Autowired
    private Environment env;

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

    @PostConstruct
    public void afterInit() {
        boolean hasDevProfile = Arrays.asList(env.getActiveProfiles()).contains("dev");
        boolean hasH2Database = Arrays.asList(env.getActiveProfiles()).contains("h2");
        String applicationName = env.getProperty("spring.application.name");
    }
}
```

Properties

@PropertySource

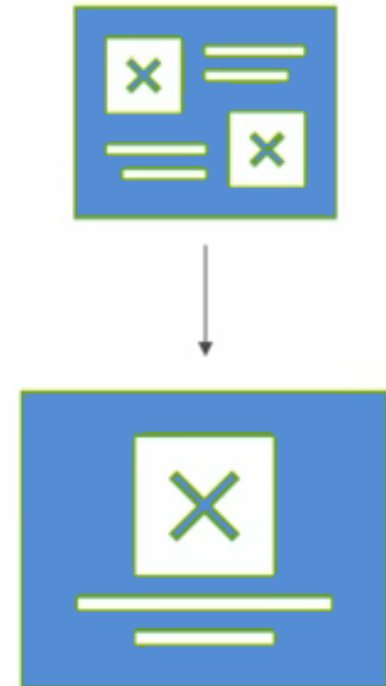
- Mit Hilfe von @PropertySource können Properties zum Environment hinzugefügt werden.

```
@Configuration
@PropertySource("classpath:jdbc.properties")
public class AppConfig {
    @Value("${jdbc.url}")
    private String jdbcUrl;
    @Bean
    public DataSource dataSource() {
        return new SimpleDataSource(jdbcUrl);
    }
}
```

Profiles

Profiles und Active Profiles

- mehrere Profile für unterschiedliche Umgebungen (Staging Umgebungen wie prod, test, dev, edu)
- Profile können den Zugriff auf Datenbanken parametrieren
- Entwicklungsprofil (z.B mit Memory Datenbank)
 - application-**dev**.properties
- Definition der aktiven Profile
 - Command Line: -Dspring.profiles.active=**dev**
 - In Properties Datei: spring.profiles.active=**dev, h2**
- Annotation `@Profile("dev")` steuert die Ausführung



Profiles

Beispiel @Profile und @Conditional

- Die Bean wird nur instanziiert bei
 - Development Profil: -Dspring.profiles.active=**dev**
 - Und falls die Klasse im ClassPath "**org.h2.Driver**" gefunden wird

```
@Configuration
@Profile("dev")
@ConditionalOnClass(name = {"org.h2.Driver"})
public class DevConfiguration {

    @Bean
    public DatabaseBootstrap databaseBootstrap() {
        return new DatabaseBootstrap();
    }

}
```

Profiles

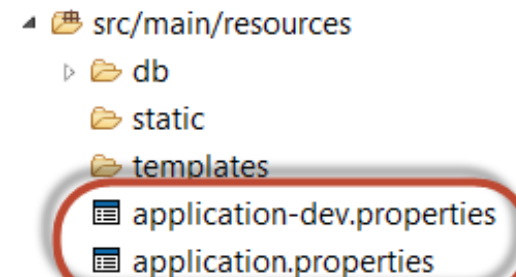
Default und dev-Properties (Beispiel)

- Default Property: application.properties

```
spring.jpa.hibernate.ddl-auto=validate  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.url=jdbc:h2:file:./database;FILE_LOCK=FS  
spring.datasource.username=sa  
spring.datasource.password=
```

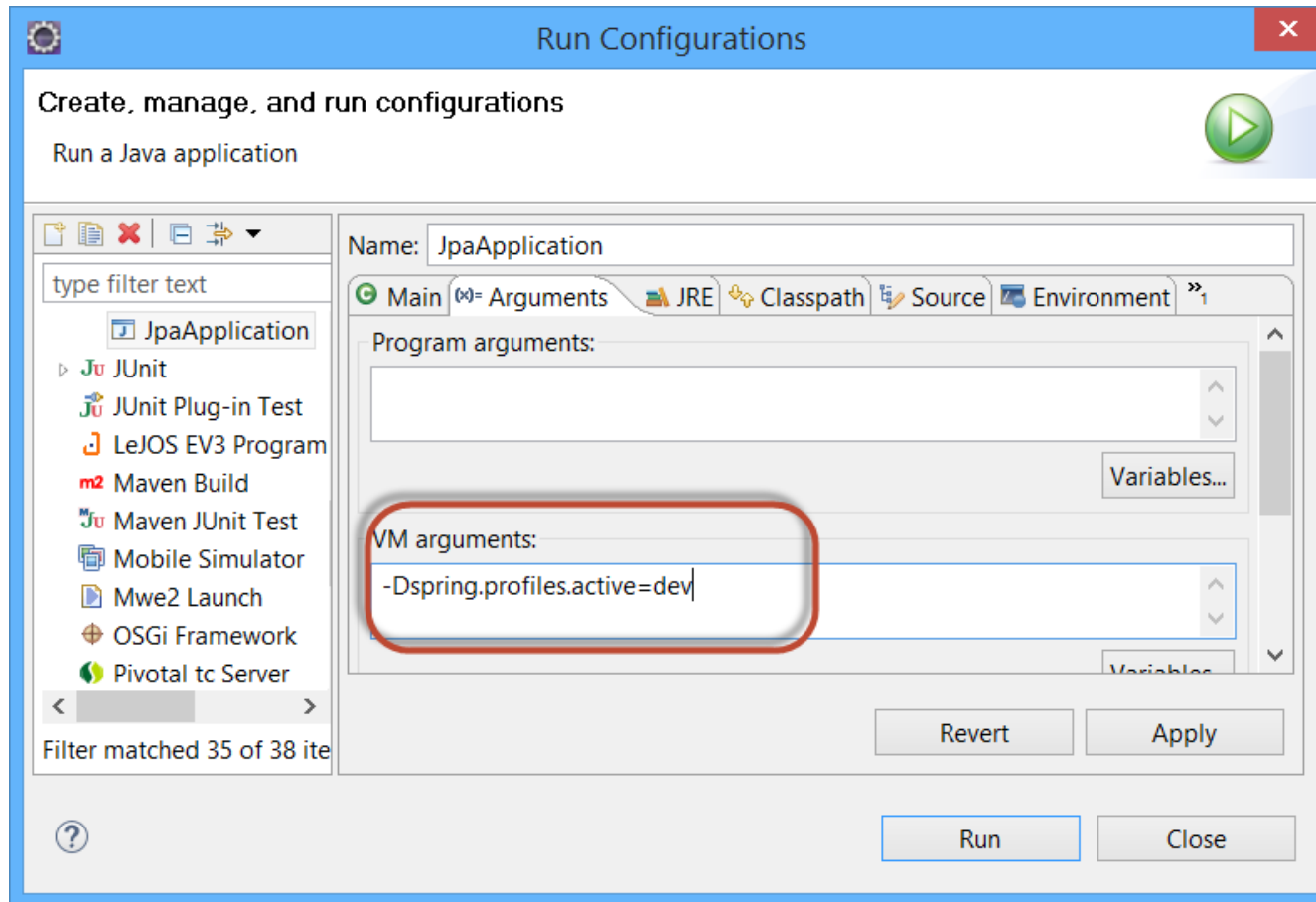
- application-dev.properties
 - H2 Konsole nur bei Entwicklungsprofil einblenden

```
spring.h2.console.enabled=true
```



Profiles

Start mit dev-Profil mit Kommandozeile



Profiles

Aktives Profil





- In Datei application.properties

```
spring.profiles.active=${ACTIVE_PROFILES:dev,h2}
```

- **ACTIVE_PROFILES** für Parametrierung über Umgebungsvariablen
- **dev,h2** sind default Werte

Hands-on

Profiles: dev, h2 und mysql

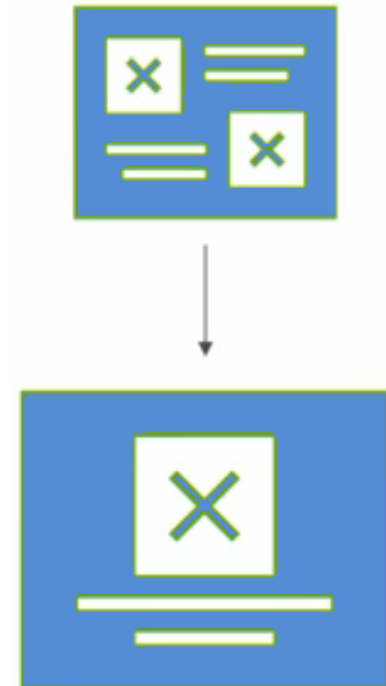
- Anleitung:
<https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest/blob/profiles/readme/liquibase.md>
- Aufteilung in verschiedene Properties Dateien
 -  application.properties
 -  application-dev.properties
 -  application-h2.properties
 -  application-mysql.properties
- Entscheidung, welche für eine Konfiguration verwendet werden:
In Datei application.properties

```
spring.profiles.active=${ACTIVE_PROFILES:dev,h2}
```


Zusammenfassung

Properties, Profile und @Conditional

- Auto Konfiguration
 - Deaktivieren von Beans mit @Conditional und @Profile
- Properties
 - Erlauben die Konfiguration des Frameworks ohne Programmierung
- Profile
 - mehrere Profile für unterschiedliche Umgebungen (Staging Umgebungen wie prod, test, dev, edu)
- ActiveProfiles
 - Legen fest, welche Profile eingesetzt werden



Deployment von Spring mit Docker

- Build
- Erstellen eines Containers
- Docker Compose Dateien

Lernziele LE 03-2 – Docker

- Die Studierenden
 - können eine FAT-JAR-Datei mittels `mvn clean package`
 - Können ein Dockerfile erstellen
 - Können mittels Docker und Docker Compose Konfigurationen in Container Infrastrukturen erstellen

Voraussetzung

- Properties für die Parametrierung mittels Umgebungsvariablen
- Build: Erstellen einer jar Datei
- Docker muss auf dem Rechner installiert sein und laufen
- Dockerfile muss vorhanden sein
- Ggf. Publikation des Docker Image nach Dockerhub
- Ggf. Konfigurationen mittels docker-compose einstellen

Build: erstellen einer Jar-Datei

- Build

```
git clone https://github.zhaw.ch/bacn/ase2-spring-boot-hellorest
cd file-system-storage git checkout docker
./mvnw clean package (windows: mvnw clean package )
java -jar target/hello-rest*.jar
```

- Im target folder entsteht eine Datei hello-rest-<version>.jar



```
target
├── classes
├── generated-sources
├── generated-test-sources
├── maven-archiver
├── maven-status
├── surefire-reports
├── test-classes
├── hello-rest-0.5.jar
└── hello-rest-0.5.jar.original
```

- Version kann in der pom Datei eingestellt werden

```
<version>0.5</version>
```

Erstellen eines Docker Images

- Dockerfile  Dockerfile

```
FROM openjdk:11-jdk-slim

ADD target/hello-rest*.jar app.jar

ARG JVM_OPTS
ENV JVM_OPTS=${JVM_OPTS}

CMD java ${JVM_OPTS} -jar app.jar
```

- Die Datei `target/hello-rest*.jar` wird als `app.jar` hinzugefügt
- Docker Image erstellen ([uportal](#) mit der eigenen docker id ersetzen)

```
docker build -t uportal/hello-rest .
```

Container testen

- Spring Boot startet mit den Default Einstellungen
(**uportal** mit der eigenen docker id ersetzen)

```
docker run -p 8080:8080 --rm -it uportal/hello-rest
```

- Container mit Ctrl c beenden

Container publizieren

- Einloggen und pushen
(**uportal** mit der eigenen docker id ersetzen)

```
$ docker login --username uportal --password  
$ docker push uportal/hello-rest
```


Docker Compose für H2

- Datei docker-compose-h2.yml
- Definition der Umgebungsvariablen

```
version: '2'

services:

  hello-rest-h2:
    image: uportal/hello-rest:latest

    restart: always
    environment:
      APP_NAME: Hello Rest with H2
      ACTIVE_PROFILES: dev,h2
    ports:
      - 8080:8080
```

Starten von docker-compose

- Starten des Containers mit log output in der Konsole

```
docker-compose -f docker-compose-h2.yml up
```

- Starten des Containers mit Detached Mode (läuft als Prozess)

```
docker-compose -f docker-compose-h2.yml up -d
```

- Löschen der Container

```
docker-compose -f docker-compose-h2.yml rm
```

Docker Compose für MySQL

- Drei Container
 - MySQL
 - Spring Boot
 - Adminer
- Adminer unter localhost:9090

Zusammenfassung

- Application.properties vorbereiten für die Parametrierung mittels Umgebungsvariablen
- Build: Erstellen einer jar-Datei
- Dockerfile erstellen
- Docker image erstellen
- Konfigurationen mittels docker-compose einstellen