An introduction to Spring Boot

Michael Simons

Version 1.0, 2017-06-30 16:50:00 CEST

Table of Contents

1. Abstract

In this workshop we'll work on a application that deals with events (like talks, conferences and so on). The scope of the application is to store those events and provide an API for retrieving, manipulating and updating events.

If we'll have some time left, we also gonna see a short demo how to orchestrate that as a microservice for a calendar application.

2. Setup

2.1. Prerequisites

You need an installed JDK, at least Java 8. If you use Git, then you can checkout the Code using Git.

The project in this workshop will be a Maven project. It contains the Maven wrapper, so you don't have to install Maven by hand if you don't have a recent copy.

An installed IDE is recommended but not necessary. Having `cURL`, a browser or, any other sophisticated HTTP client installed is beneficial for issuing HTTP requests.

As Spring Boot Project is a plain Gradle or Maven Project in the end, all three major IDEs (NetBeans, Eclipse / Spring Tool Suite and IntelliJ IDEA) can open and compile them. Additional support like code completion for configuration files, request mappings, dashboards and so on vary from IDE to IDE. You have to decide what you want and need.

If you want to try out the microservice examples in Docker, you'll need a recent Docker installation on your machine, too.

If you don't have docker installed, the projects won't build after commit #445fcec. I have tagged the latest version that works without docker for you, so you can check it out with `git checkout no-docker`.

Downloads

- Oracle JDK, please choose your operating system
- Git
- One of those:

    - Spring Tool Suite
    - NetBeans with NB-SpringBoot
    - IntelliJ IDEA
- Docker Community Edition, please choose your operating system

2.2. Register a GitHub account (Optional, recommended)

Go to https://github.com and signup for a free account. If you already have a GitHub account, you can skip this step.

2.3. Clone the repository

The code of this project will be available at GitHub at https://github.com/michael-simons/ws-20170627-cluj after the workshop.

2.4. Prepare your IDE

Choose one of

- Spring Tool Suite (STS) from https://spring.io/tools
- NetBeans from https://netbeans.org
- Eclipse from https://www.eclipse.org
- IntelliJ IDEA from https://www.jetbrains.com/idea/

Even though Spring Boot projects are just regular Maven Projects, you'll get some perks from IDEs, for example direct integration with Spring Boots initializer, recognition of Spring Boots configuration, Spring Beans and more.

STS and IntelliJ offer the most complete support, however in the case of IntelliJ only the ultimate edition does.

STS is completely free. You can get its integration into Eclipse as well from the marketplace, but I'll recommend using STS directly.

NetBeans is one of the best choices for Maven projects. It also features direct support of project Lombok for example. If you're a NetBeans user, I'll highly recommend to install *NB-SpringBoot* from the plugin page or directly from NetBeans through **Tools › Plugins › Available Plugins**. Find more information on the plugin here: https://github.com/AlexFalappa/nb-springboot.

We'll discover some features of NetBeans, STS and IntelliJ on the way through the workshop, but let me assure you once again: Spring Boot projects are just plain normal Maven or Gradle projects.

3. Create a new Spring Boot project

In this part you'll learn about the different ways to bootstrap a Spring Boot application.

As you have learned in the introduction, Spring Boot projects are basically Spring projects that just happend to configure stuff in a way that seems "magically" at first look. This configuration comes in the form of a parent project in the case of Maven and in the form of a plugin if you're a Gradle user.

You can write the build file by hand or you just can go to start.spring.io and have the Spring Initializr generate it for you.

3.1. Use the Spring Initializr

Through the website

Steps

1. Go to https://start.spring.io, chose wether you want to use Maven or Gradle, the language (in this workshop we'll use Java) and the version of Spring Boot (leave as is).
2. Enter "Web" inside dependencies textbox and select "Web"
3. Repeat last steps with the technologies you'll find interesting
4. Hit generate

Your browser downloads a Zipfile with your project. You can unzip this with a tool of your choice and already start your application:

Example 1. Unzip and start the first demo

```
unzip demo.zip
cd demo
./mvnw spring-boot:run
```

To explore the generated buildfile and the minimal application skeleton generated, we're repeat the steps inside an IDE.

From an IDE

Spring Tools Suite

Steps

- Choose **File › New › Spring Starter Project**
- Enter `event-service` as a name

- Enter `ac.simons.ws.cluj:events` as coordinates (feel free to chose whatever you like, though. However: this document and the finished project will refer to those coordinates)
- Enter some appropriate description and name
- Clear the package name
- Hit next and chose "Web" as dependency
- Hit either finish or next. Next gives you a handy URL that can be used to reference exactly the options you used

Wait a second until STS automatically opens your project.

NetBeans

The steps are basically the same, but you have to have the NB-SpringBoot-Plugin installed.

- Choose **File › New Project… › Maven › Spring Initializr Project**
- Enter your coordinates and description as described above
- Choose dependencies as described before
- Select a location
- Uncheck *Remove Maven Wrapper* if you want to keep the Maven Wrapper inside the generated project (which is useful, if you ask me)

You're done.

You'll notice that neither STS nor NetBeans store a lot of IDE specific stuff. Both have some project specific settings, like which is the main class and so on, but that's basically it. You can run both projects from the command line.

IntelliJ IDEA Ultimate Edition

Again, the same steps. Hit "Create new project", choose "Spring Initializr" and follow the dialog.

With cURL or other tools

The Spring Initializr is completely scriptable. It speaks Hypertext Application Language (HAL) and you can get an overview about its options by just curling it:

Example 2. Retrive the metadata of Spring Initializr

```
curl -H 'Accept: application/json' https://start.spring.io
```

Find a complete documentation at docs.spring.io/initializr. The example from the beginning of this chapter also be created via

Example 3. Use Spring Initializr from the command line

```
curl https://start.spring.io/starter.zip -d dependencies=web -o demo.zip
unzip demo.zip -d demo
cd demo
./mvnw spring-boot:run
```

Explore the manual linked above. You can install a custom copy of the Initializr if you want to. Scripting it might be a valuable tool for your process.

4. Explore your first Spring Boot project

If you didn't follow the steps above, now it's time to open the project `event-service`. Navigate to the folder that you used while cloning the repository and open the project with your IDE of choice.

4.1. The build file

First check out the `pom.xml`. Note that it defines a parent through

Example 4. Standard Spring Boot projects inherited from Spring Boots parent pom

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.4.RELEASE</version>
    <relativePath/>
</parent>
```

Then follow some properties. Have a look at `java.version`. Its set to 1.8 by default. You don't have to configure Mavens compiler plugin. The parent does that for you.

Spring Boot prior to 2 supports Java 7 and 8 (Java 6 with some workarounds), Spring Boot 2 needs Java 8 and will support Java 9.

Build-in dependency management

Then follow the dependencies, which we just declared:

Example 5. Some Spring Boot dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

Noticeable here is the lack of version numbers. That has been taken care of by the Spring and Spring Boot teams: They chose versions of libraries that work well together and put them in their parent pom in a section `<dependencyManagement />` Now everytime you need one of those, you can skip the version number.

If you want to override a version, you can do that too via a property declared like the Java version. Later in the example we're gonna use Flyway and declare it like so

Example 6. Another dependency without explicit version

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
```

If you want to use a version other than Spring Boot uses, you wouldn't overwrite it in the dependency but declare a property:

Example 7. Overwriting versions via properties

```
<properties>
    <flyway.version>4.2.0</flyway.version>
</properties>
```

"Starter"

What the hell are starter? Those are "meta-dependencies". They usually consist of an autoconfigure module that has code for configuring certain aspects of a framework module or library. Usually that autoconfigure module depends only optional on the framework module or library. The starter module itself depends on the autoconfiguration as well as on the libraries.

4.2. The main class

Depending on how you parameterized the initializer, your main class will be named differently. In the example project it is `EventServiceApplication`, named after the coordinates you entered and located in the package, which you entered explicitly. The class looks pretty innocent:

Example 8. A default Spring Boot main class

```
package ac.simons.ws.cluj.events;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EventServiceApplication {

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

The main method is interesting, because it delegates to a static helper method inside `SpringApplication` that does all the heavy lifting of initializing the Spring context.

But where does the configuration come from? Explore the `@SpringBootApplication` annotation. It's a composed annotation that looks like this:

Example 9. Abbreviated source of `@SpringBootApplication`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication{}
```

This annotation combines several others:

- `@SpringBootConfiguration` marks the class as a configuration class. `@SpringBootConfiguration` is special in so far that there should be only one of those throughout one context whereas `@Configuration` classes can be many
- `@EnableAutoConfiguration` enables the "magic" of Spring Boot: It looks for `@Configuration` classes that are marked as autoconfiguration and loads them fully or partially, depending on wether certain conditions are fulfilled or not.
- `@ComponentScan` finally kicks of the search for Spring components. Those are `@Controller`, `@Services` and many others

As you can see, the application class has a main method and therefor can be used for a runnable jar file. Part of Spring Boots philosophy is to provide single fat jars as deployment artifacts.

As we have declared the WEB dependency and got the `spring-boot-starter-web`, we also have an embedded Tomcat on the classpath.

However, you can also choose to deploy Spring Boot applications as war. If you had chose 'WAR' as packaging type in the initializer, it would have created an additional `SpringBootServletInitializer` that facilitates Springs SPI for `ServletContainerInitializer`.

How does that work with the embedded tomcat? There are Maven and Gradle plugins that repackages the artifact to be a "fat jar" or "fat war". The "main" class we have written isn't directly called, but a Spring Boot loader class.

4.3. The package structure

At the moment, there's only one class, the application class. The package is based on the project coordinates. The package structure is actually already important here as the application class is annotated with `@ComponentScan`. This annotations searches for Spring components from the package the annotated class declares downwards.

There are two caveats:

- Don't annotate a class in the root package with `@SpringBootApplication` or `@ComponentScan`. It will scan the whole class path!
- Those annotation won't scan packages in parallel to their current package

How about the configuration in starter than? Does Spring actually run a full classpath package scan? No: It uses the `spring.factories` services locator implementation!

4.4. The test sources

The initializer already generated a test:

Example 10. Generated test

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class EventServiceApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

Its a standard JUnit 4 test that is run with a special runner, the `SpringRunner` and marked as `@SpringBootTest`. The later denotes a full integration test: Loading the embedded web container (if any) and all connections to third party services.

5. Add some content

5.1. Spring Boot Developer Tools

Let's add a nice runtime dependency call "Spring Boot Developer Tools", "devtools" for short:

Example 11. Declare Spring Boot devtools

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
</dependency>
```

This is a not a compile time but a runtime dependency that has several neat features:

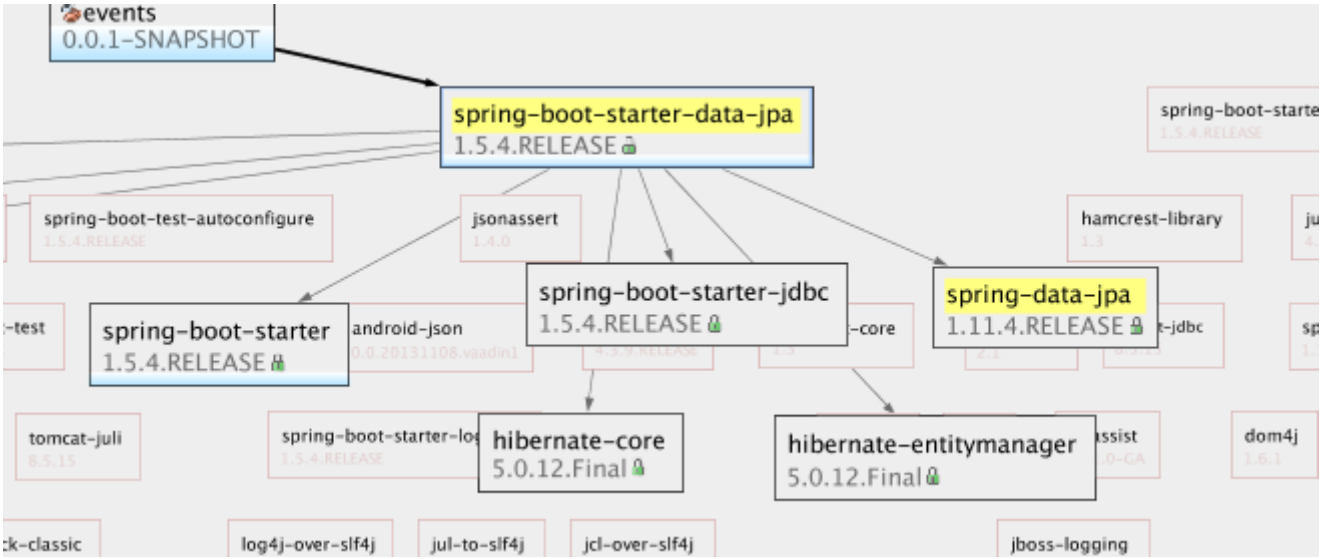
- When the application is run from an IDE or with the Maven or Gradle plugin, it restarts the context when classes changes
- It automatically reloads changed resources
- Changes some settings during development, for example disables caching for messages, templates and so on

It's not as elaborate like JRebel, but, nevertheless, valuable!

5.2. Spring Boot Starter JPA

We have to store some stuff. `org.springframework.boot:spring-boot-starter-data-jpa` is a handy starter that brings you among others:

- Hibernate Core
- Hibernate Entity Manager
- Spring JDBC
- Spring Data Commons
- Spring Data JPA



We gonna deep dive into this later. First step is to declare a simple entity for storing events:

Example 12. Simple JPA entity

```
@Entity
@Table(
        name = "events",
        uniqueConstraints = {
            @UniqueConstraint(name = "events_uk", columnNames = {"held_on",
"name"})
        }
)
public class EventEntity implements Serializable {

    private static final long serialVersionUID = 2005305860095134425L;

    public enum Status {

        open, closed
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "held_on", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar heldOn;
```

```
    @Column(length = 512, nullable = false)
    private String name;

    @Column(name = "created_at", nullable = false)
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar createdAt;

    @Enumerated(EnumType.STRING)
    private Status status;

    protected EventEntity() {
    }

    public EventEntity(final Calendar heldOn, final String name) {
        this.heldOn = heldOn;
        this.name = name;
        this.status = Status.open;
    }

    @PrePersist
    @PreUpdate
    void prePersistAndUpdate() {
        if (this.createdAt == null) {
            this.createdAt = Calendar.getInstance();
        }
    }
}
```

As you can see: Nothing apart from Hibernate specific annotations and nothing fancy here.

Using Spring with Spring Boot and the JPA starter you don't have to worry about a persistence unit. Spring Boot takes care of

- Collecting all entity classes and related classes
- Provides a datasource
- Provides local transaction management for that datasource
- Provides an EntityManagerFactory
- Provides a thread safe entity manager, independent wether JTA or application based transactions are used → You can omit `@PersistenceContext` annotation which works on attributes only

5.3. Web layer

We already have all the dependencies we need to work on the web layer, so we can add a controller that should take care of handling events:

Example 13. Empty Spring Web MVC controller

```
@RestController
public class EventApi {
}
```

Nothing there yet, we're gonna develop the controller in a test driven way.

Inside your test sources create a class `EventApiTest` in the same package as your controller having the following content:

Example 14. Test the controller above

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class EventApiTest {
    @Test
    public void getEventsShouldWork() {
        // Actually test something ;)
    }
}
```

What do we have here: Again a Spring JUnit Test, but this time `@WebMvcTest`. This annotation is called a *test slice*. It only configures infrastructure and Spring components essential for that technical slice. In this case: The web layer.

`@WebMvcTest` first scans the package of the test class for a context configuration. If it doesn't find a class with `@SpringBootApplication` it uses the same rules as the Spring Framework itself, i.e. it looks also for XML and Groovy based configuration. If that isn't successful, it also scans the class path "upwards", so it finds your main Spring Boot class even if you have put away your controllers inside a subpackage.

Example 15. First iteration of the test

```
@Autowired
private MockMvc mvc;

@Test
public void getEventsShouldWork() throws Exception {
    this.mvc
            .perform(MockMvcRequestBuilders.get("/api/events"))
            .andExpect(MockMvcResultMatchers.status().isOk());
}
```

That first approach uses an instance of `MockMvc` provided by `@WebMvcTest` to call the events api and expects a status 200 (ok). That test obviously fails. Let's make this work by adding one method to the controller:

Example 16. Fix the failing test

```
@GetMapping("/api/events")
public List<EventEntity> getEvents() {
    return new ArrayList<>();
}
```

Now run the test again and see it turn green!

We can easily break it again by actually checking the result:

Example 17. Break the test again!

```
@Test
public void getEventsShouldWork() throws Exception {
    this.mvc
            .perform(MockMvcRequestBuilders.get("/api/events"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$[0].name",
equalTo("test1")));
}
```

This one checks if the returned content is actually valid. Naive implementation would be hitting the entity manager from within the controller but how could we test just the controller and keeping away from the database? Let's introduce a service:

Example 18. Simple service that does the heavy lifting for us

```
@Service
public class EventService {
    public List<EventEntity> allEvents() {
        return new ArrayList<>();
    }
}
```

This class is picked up by the Spring context and can be injected into other beans. Lets cleanup the controller in the way:

Example 19. Revamped controller

```
@RestController
@RequestMapping("/api/events")
public class EventApi {
    private final EventService eventService;

    public EventApi(EventService eventService) {
        this.eventService = eventService;
    }

    @GetMapping
    public List<EventEntity> getEvents() {
        return this.eventService.allEvents();
    }
}
```

You see only annotations relevant to Spring Web MVC. Try running the test know: It doesn't even start any more: The Service is not part of the webslice! Spring Boot actually helps you a lot here with its failure analyzers:

Example 20. Failure analysis

```
Error starting ApplicationContext. To display the auto-configuration report re-run
your application with 'debug' enabled.
2017-06-19 12:07:57.297 ERROR 73221 --- [           main]
o.s.b.d.LoggingFailureAnalysisReporter   :

***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of constructor in ac.simons.ws.cluj.events.EventApi required a bean of
type 'ac.simons.ws.cluj.events.EventService' that could not be found.


Action:

Consider defining a bean of type 'ac.simons.ws.cluj.events.EventService' in your
configuration.
```

Instead of providing the real bean, we're gonna use `@MockBean`:

Example 21. Providing collaborateurs through `@MockBean`

```
@RunWith(SpringRunner.class)
@WebMvcTest
public class EventApiTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EventService eventService;
}
```

That would be possible on class level, too, but we're gonna need that been together with the Mockito support that the dependency on `org.springframework.boot:spring-boot-starter-test` brought:

Example 22. Prepare the mock

```
ZonedDateTime now = ZonedDateTime.now();
        List<EventEntity> expectedEvents
                = Arrays.asList(
                        new EventEntity(GregorianCalendar.from(now.plusDays(3)),
"test1"),
                        new EventEntity(GregorianCalendar.from(now.plusWeeks(1)),
"test2")
                );
        Mockito.when(eventService.allEvents()).thenReturn(expectedEvents);
```

5.4. Database layer

Now that we have tested the web layer, we'll move back again to the database. Together with the Spring Data JPA dependencies we have added `com.h2database:h2` as a runtime dependencies. Spring Boot configures an in-memory instance of H2 if no other database connection is configured. So, we already have a datasource.

Where does the schema come from? Spring Boot configures `spring.jpa.hibernate.ddl-auto` for you: It uses `create-drop` on an embedded database, `none` otherwise which is a sensible default.

Spring Boot also takes `schema.sql` and `data.sql` scripts in the root of the classpath in consideration. If there is a `schema.sql` Spring uses that for database initialization **before** `data.sql` **and** before JPA. If there's only `data.sql`, Spring Boot uses first JPA to generate schema and than the script.

In the example we're gonna use `data.sql` only and let Hibernate generate the schema for us.

First of all we're gonna rework the generated test. By default it's only mocking the web

environment. Let's start in on a random port through `@SpringBootTest(webEnvironment = RANDOM_PORT)`. That gives also a `TestRestTemplate` that makes calling our a API a breeze:

Example 23. Test first, again

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = RANDOM_PORT)
public class EventServiceApplicationTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void getEventsShouldWork() {
        final List<EventEntity> events = this.restTemplate.exchange(
                "/api/events",
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<List<EventEntity>>() {}
        ).getBody();
        assertThat(
                events.get(0).getName(),
                is(equalTo("Get the most out of your data layer"))
        );
    }
}
```

The test fails as expected, but safely assuming we do have some content now, let's fill the service with life and make the test work:

Example 24. Event service based on plain JPA database access

```
@Service
public class EventService {
    private final EntityManager entityManager;

    public EventService(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Transactional(readOnly = true)
    public List<EventEntity> allEvents() {
        return this.entityManager
                .createQuery(
                        "Select event from EventEntity event order by
event.heldOn",
                        EventEntity.class
                )
                .getResultList();
    }
}
```

And green again!

Using a repository abstraction

Your service should have to deal with the persistence storage on it's own, meaning, it should have no knowledge of the underlaying technology.

The repository pattern fixes that. Implemented in Spring Data JPA it takes away the burden of interacting with the persistence layer in many cases.

The abstraction is in so far leaking that you get JPA entites out of a JPA repository and Mongo documents out of a Mongo repository. It's up to you to encapsulate this further.

We have several tests in place and can try to rework our service.

Spring Boot supports Spring Data JPA out of the box and you can declare a repository like this:

Example 25. Spring Data JPA repository for the Events

```
public interface EventRepository extends Repository<EventEntity, Integer> {
    List<EventEntity> findAllByOrderByHeldOnAsc();
}
```

If you already know Spring Data you'll notice that I don't used the `JpaRepository` interface as I prefer only to declare the methods I actually need and use.

The rewritten service based on that repository now looks like this:

Example 26. Rewritten event service

```
@Service
public class EventService {
    private final EventRepository eventRepository;

    public EventService(EventRepository eventRepository) {
        this.eventRepository = eventRepository;
    }

    public List<EventEntity> allEvents() {
        return this.eventRepository.findAllByOrderByHeldOnAsc();
    }
}
```

And our test is still green.

Testing the service

We have several ways of testing the service. We can either mock the repository (which I usually do), or we can use yet another test slice, that is: `@DataJpaTest`. `@DataJpaTest` replaces databases via an embedded Database by default, runs all init scripts or migrations and then runs a transactional test. Transactional tests are rolled back by default:

Other options to initialize a database

Although you can use several `schema.sql` and `data.sql` scripts and also relay on JPA to generate your tables, I'm very sceptic about that. Especially the JPA based migrations work only well if you're and your application is in charge of the database. That is: Has the right to change schema at will.

The project contains therefor contains a dependency to Flyway (`org.flywaydb:flyway-core`) core, that takes care of using dedicated scripts to initialize your database. An alternative would be using Liquibase.

6. Configuration of Spring Boot Applications

You can follow several paradigms to configure a Spring Boot application. You're either can relay completely on the environment and let your Spring Boot application adapt itself or you chose profiles or combinations thereof. Either way: In no case you have to build your applications for different environments differently.

You should differentiate between external and internal configuration. We speak about internal configuration in regards of beans, context- and dependency injection and so on. External configuration on the other hand basically describes means to change the behavior of your application depending on environment or configuration properties.

Internal configuration often depends on external configuration!

6.1. External configuration

External configuration come from a so called `PropertySource`. Those property sources can be of various kind and have a well defined order:

- `@TestPropertySource` annotations on your tests.
- `@SpringBootTest#properties` annotation attribute on your tests.
- Command line arguments
- Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment
- variable or system property)
- `ServletConfig` init parameters
- `ServletContext` init parameters
- JNDI attributes from `java:comp/env`
- Java System properties (`System.getProperties()`)
- OS environment variables
- A `RandomValuePropertySource` that only has properties in `random.*`
- Profile-specific application properties outside of your packaged jar
- (`application-{profile}.properties` and YAML variants)
- Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants)
- Application properties outside of your packaged jar (`application.properties` and YAML variants)
- Application properties packaged inside your jar (`application.properties` and YAML variants)
- `@PropertySource` annotations on your `@Configuration` classes.
- Default properties (specified using `SpringApplication.setDefaultProperties`)

Now, open the file `application.properties`. It corresponds to the `default` profile and configuration of your application.

If you're a YAML fan, you can use YAML as well.

All IDEs mentioned here support syntax highlighting and auto completion in this file.

How to use those properties?

You basically have to options:

1. The `@Value` annotation that can be used to retrieve any property from the environment.
2. Classes annotated with `@ConfigurationProperties`

The second option has several advantages:

- You can concentrate configuration for one topic in one class, including the possibility to provide defaults
- The class are subject to relaxed binding
- They are available as beans in the context
- They are recognized by a build processor to generate meta data which in turn is helpful for content assist and other

Given the following pretty arbitrary properties bean:

Example 27. Some arbitrary properties

```
@Component
@ConfigurationProperties(prefix = "event-service")
public class EventServiceProperties {
    /**
     * The default number of seats avaiable for each event.
     */
    private Integer defaultNumberOfSeats;

    /**
     * Some arbitrary information.
     */
    private String arbitraryInformation;

    public Integer getDefaultNumberOfSeats() {
        return defaultNumberOfSeats;
    }

    public void setDefaultNumberOfSeats(Integer defaultNumberOfSeats) {
        this.defaultNumberOfSeats = defaultNumberOfSeats;
    }

    public String getArbitraryInformation() {
        return arbitraryInformation;
    }

    public void setArbitraryInformation(String arbitraryInformation) {
        this.arbitraryInformation = arbitraryInformation;
    }
}
```

The default number of seats can be configured by any of those:

- `event-service.default-number-of-seats = 42`
- `eventService.defaultNumberOfSeats = 42`

- `event_service.default_number_of_seats = 42`

Chose the format that fits the source best: Usually uppercase and underscores works great in environment properties, dashes are good for properties.

To provide metadata of this configuration file for you, you're IDE and your coworkers add the `spring-boot-configuration-processor`

Example 28. Use the spring-boot-configuration-processor

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

7. Non functional requirements

Some non functional requirements that usually are requested:

- Health information
- Metrics
- Context and configuration information
- Logging

Spring Boot offers Spring Boot Actuator that can be added as simple dependency:

Example 29. Add Spring Boot Actuator

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

This provides a several interesting new api endpoints, including the information above.

With current Spring Boot they reside directly under the root context:

- http://localhost:8080/health
- http://localhost:8080/metrics
- http://localhost:8080/info
- http://localhost:8080/autoconfig

Spring Boot will change that to `/application/` and you can configure that already today. And while

we're at it, disable actuator endpoint security as the endpoints are protected by default and since we don't have Spring Security yet, there's no valid user to authenticate with:

Example 30. Configure Spring Boot actuator

```
management.security.enabled = false
# That will be the default with Spring Boot 2.0
management.context-path = /application
```

8. Microservice Scenario

The sources contain one microservice scenario which makes use of two projects from the Spring Cloud Netflix portfolio:

- It uses Eureka for Service Discovery: Eureka instances can be registered and clients can discover the instances using Spring-managed beans
- It uses Hystrix as a service breaker: Hystrix clients can be built with a simple annotation-driven method decorator

This is just a demonstration of how one can use those projects. It should not by any means recommend this kind of microservice architecture for all use cases. Its a nice and easy way to demonstrate what you can do with additional Spring projects after finishing your Spring Boot application.

The project `calendar-service` is a client of `event-service`. In addition there is now `service-registry`.

`service-registry` declares an additional dependency management:

Example 31. Bring in Spring Cloud dependencies

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
</dependencies>
```

And the application is annotated like this:

Example 32. Provide an Eureka server:

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceRegistryApplication {

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

Thus the Spring Boot application boots the Eureka instance. In addition it tries to register itself with other instances. As it's the only one, disable this:

Example 33. Disable registration with other registry

```
server.port = 8761

eureka.client.register-with-eureka = false
eureka.client.fetch-registry = false
```

The listing above also sets Springs server port to Eurekas default. You build and run the service registry as usual. If you do this, you'll reach a admin interface at http://localhost:8761.

Then back to the event service. Add the same additional dependency management as well as `org.springframework.cloud:spring-cloud-starter-eureka-server`.

Also, give the service a fixed name through configuration:

Example 34. Fixed name for a Spring Boot application

```
spring.application.name = event-service
```

You'll now annotate the `EventServiceApplication` with `@EnableDiscoveryClient`

Example 35. Register a service with the service registry

```
@SpringBootApplication
@EnableDiscoveryClient
public class EventServiceApplication {

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

`@EnableDiscoveryClient` enables discovery for Eureka instances by default, but there are also implementations for Zookeeper and other.

Imagine the `event-service` as provider. The additional `calendar-service` in this repository is a client. The `calendar-service` should retrieve all existing events and present them in a calendar widget. It's main application class shall also be annotated with `@EnableDiscoveryClient`.

Both services are now registered with Eureka.

We'll use Springs `RestTemplate` to synchronously `GET /api/events` from our event service, but we won't be hardcoding the server path and port of the event service. Instead, we'll provide a special instance of `RestTemplate` through the following configuration inside `calendar-service`:

Example 36. Provide a load balanced RestTemplate

```
@Configuration
public class RestTemplateConfig {

    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

You see an ordinary configuration class that returns one bean. The method is annotated with `@LoadBalanced`. `@LoadBalanced` also comes with `spring-cloud-starter-eureka-server` and is part of Springs so called Ribbon client. Ribbon is a client side load balancer that knows the connection to the service discovery. If you use `@EnableDiscoveryClient` with your application and are ok with Ribbons default, you can use it immediately to make your `RestTemplate` aware of virtual host names.

Virtual host names are the names of the applications that are registered with the Eureka service registry instance.

The only difference in using the template is the fact, that you have to provide the name of a virtual host instead of a server and port:

Example 37. Using a load balanced RestTemplate

```
@Service
public class EventService {
    public List<Event> allEvents() {
        final ResponseEntity<List<Event>> response =  this.restTemplate
                .exchange("http://event-service/api/events", HttpMethod.GET, null,
new ParameterizedTypeReference<List<Event>>() {
                });
        if(response.getStatusCode() != HttpStatus.OK) {
            throw new RuntimeException("Could not retrieve events");
        }
        return response.getBody();
    }
}
```

And that's all! As you see, we have wrapped the call inside a service to which we'll come back later.

Using the service from with in a controller like this posses a problem though:

Example 38. Using the service above

```
@Controller
public class IndexController {

    private final EventService eventService;

    public IndexController(EventService eventService) {
        this.eventService = eventService;
    }

    @GetMapping({"/", "index"})
    public String index(final Model model) {
        model.addAttribute("events", this.eventService.allEvents());
        return "index";
    }
}
```

The request against the controller depends on another request from server to server that can fail (and usually will fail some times).

There are many different solutions for that topic. One is a technical solution called "circuit breaker", much like a fuse inside your house that opens when an error happens and depending on the fuse will close again, either manually or after some time, trying again.

From the Netflix stack there's `org.springframework.cloud:spring-cloud-starter-hystrix` that provides `@HystrixCommand`. To use those circuit breakers inside your application, they have to be enabled like:

Example 39. Enable circuit breaker

```
@SpringBootApplication
@EnableCircuitBreaker
public class CalendarServiceApplication {

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

And then it's easy to use them. In this case we're using the `fallbackMethod` attribute to define another method that is to be called when the original method fails for some reason:

Example 40. Using a circuit breaker

```
@HystrixCommand(fallbackMethod = "emptyEvents")
public List<Event> allEvents() {
    // As above make call to remote server
    return response.getBody();
}

List<Event> emptyEvents() {
    return new ArrayList<>();
}
```

Depending on other settings, Hystrix will keep the circuit open for some time, than close and try again.

Take care in a production system to adapt Hystrix timeouts to Ribbon clients timeouts.

That may works when a service depends on one remote call, but having different calls, it gets hard very fast to manage. A better solution for that may be architectures that invert the relation ship: Whereas we'll keep the events as a resource, that resources is polled in some sane interval and events are stored redundant in the calendar service.

Changes from the calendar service to events can be communicated maybe through an event bus.

9. Wrap up

We have managed to setup one of many possible micro service architectures in about 4 hours.

You can run them independent now each as explained in the beginning and than hit http://localhost:8088 for the calendar service or use the provided docker images as follows (given a valid Docker installation on your machine)

Build everything with either `build.sh` or `build.bat`. That will use Docker Maven Plugin to provide three images:

- msimons/ws-cluj-calendar-service
- msimons/ws-cluj-event-service
- msimons/ws-cluj-service-registry

Which are orchestrated via the following `docker-compose.yml`

Example 41. Orchestrate docker images via docker-compose

```
version: '2'

services:
  calendar-service:
    image: msimons/ws-cluj-calendar-service
    ports:
      - "8080:8080"
    depends_on:
      - service-registry
      - event-service
    environment:
      - server.port=8080
      - eureka.client.serviceUrl.defaultZone=http://service-registry:8761/eureka/

  event-service:
    image: msimons/ws-cluj-event-service
    depends_on:
      - service-registry
      - database
    environment:
      - server.port=8080
      - spring.profiles.active=prod
      - spring.datasource.url=jdbc:postgresql://database:5432/events-db
      - eureka.client.serviceUrl.defaultZone=http://service-registry:8761/eureka/
    expose:
      - "8080"

  service-registry:
    image: msimons/ws-cluj-service-registry
    expose:
     - "8761"

  database:
      image: postgres:9.6
      environment:
          - POSTGRES_USER=events-db
          - POSTGRESS_PASSWORD=events-db
```

Notice how some properties of your applications in the containers can easily changed through dockers environment setting. For example we have to configure Eurekas default zone now with `eureka.client.serviceUrl.defaultZone` as Eureka doesn't run on the same host anymore.

Run everything now with `docker-compose up` and the calendar service is reachable as a frontend for the whole system under http://localhost:8080.