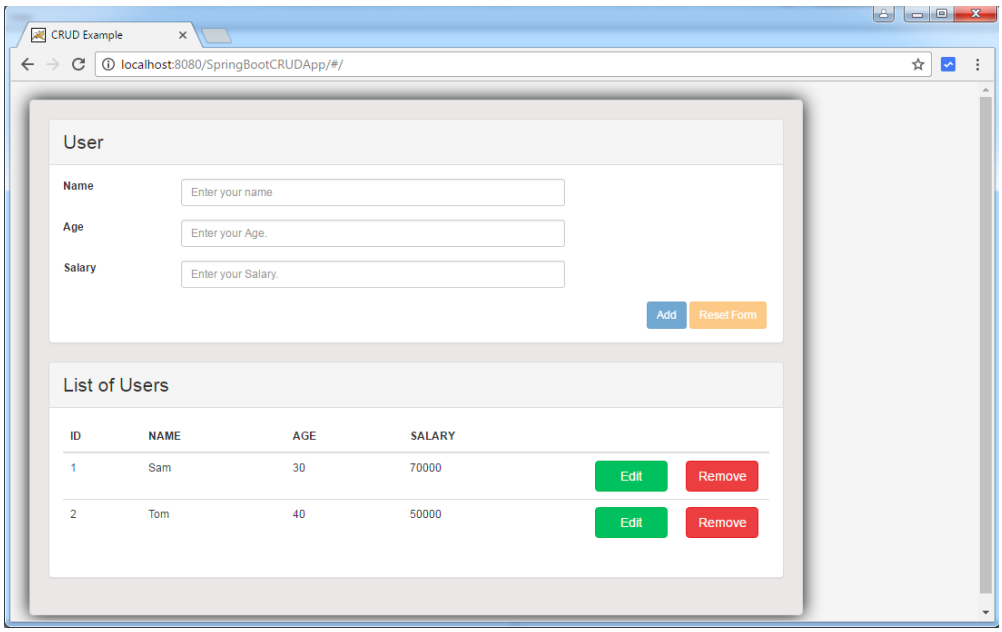
Spring Boot + AngularJS + Spring Data + JPA CRUD App Example

Created on: December 27, 2016 | **Last updated on:** March 11, 2017 websystiqueadmin

In this post we will be developing a full-blown CRUD application using `Spring Boot`, `AngularJS`, `Spring Data`, `JPA/Hibernate` and `MySQL`, learning the concepts in details along the way. This application can as well serve as a base/starting point for your own application. In addition, we will also use the notion of profiles to deploy the application into two different databases [H2 & MySQL] to emulate the local and production environment, to be more realistic.

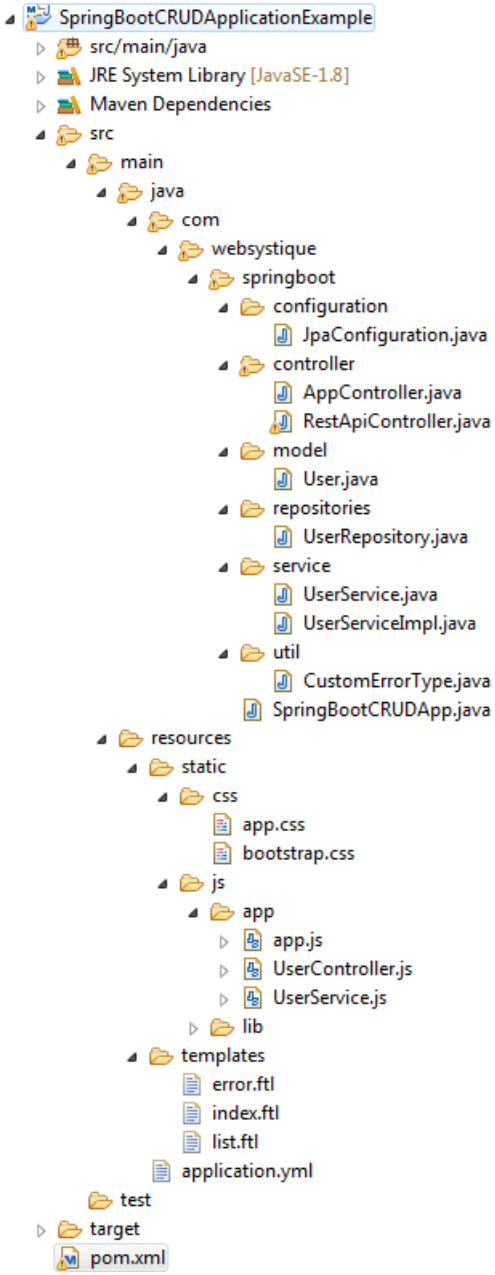


Following technologies stack being used:

- Spring Boot 1.4.3.RELEASE
- Spring 4.3.5.RELEASE [transitively]
- Spring data JPA 1.10.6.RELEASE [transitively]
- Hibernate 5.0.11.Final [transitively]
- MySQL 5.1.40 [transitively]
- H2 1.4.187
- Hikari CP 2.4.7 [transitively]
- AngularJS 1.5.8
- Maven 3.1
- JDK 1.8
- Eclipse MARS.1

Let's Begin.

1. Project Structure



2. Dependency Management [pom.xml]

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.o
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.websystique.springboot</groupId>
    <artifactId>SpringBootCRUDApplicationExample</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <name>SpringBootCRUDApplicationExample</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.3.RELEASE</version>
    </parent>

    <properties>
        <java.version>1.8</java.version>
        <h2.version>1.4.187</h2.version>
    </properties>

    <dependencies>
        <!-- Add typical dependencies for a web application -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Add freemarker template support -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-freemarker</artifactId>
        </dependency>
        <!-- Add JPA support -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- Add Hikari Connection Pooling support -->
        <dependency>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
        </dependency>
        <!-- Add H2 database support [for running with local profile] -->
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>${h2.version}</version>
        </dependency>
        <!-- Add MySQL database support [for running with PRODUCTION profile]
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.5</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin><!-- Include if you want to make an executable jar[FAT JA
                includes all dependencies along with sprinboot loader] that y
                    commandline using java -jar NAME -->
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

`spring-boot-starter-parent` : In most of the cases[unless imported], your maven project POM will simply inherit from the spring-boot-starter-parent project. The spring-boot-starter-parent provides useful Maven defaults, provides dependency-management section so that you can omit version tags for dependencies you would need for your own project. Once inherited from spring-boot-starter-parent, you would declare dependencies to one or more “Starters” jars.

`spring-boot-starter-web` : Provides typical WEB MVC + Embedded container support.

`spring-boot-starter-freemarker` : Provides freemarker template support. We will be using freemarker in this example.

`spring-boot-starter-data-jpa` : Provides spring-data setup using JPA abstraction. Since we are talking about fast-development using spring-boot, spring-data would certainly save time compare to traditional DAO/Creteria/Query manual setup.

`HikariCP` : Provides Hikari connection pooling support. We could have as well used Tomcat datapooling. Common DBCP is usually not recommended for performance reasons.

`h2`: Provides H2 database support. Please note that it is used here just to demonstrate the real-life scenarios where your local setup uses one database while the one on production might be altogether a different database.Additionally, we are deliberately using a different version of h2, just to demonstrate that you CAN change the dependencies if needed.

`mysql-connector-java`: Provides MySQL database support. Again, just because we are simulating a local[H2]-Production[MySQL] scenario.

2. Spring Boot Application [Main class]

You read it right. Good old main is what all we need to start our newly created spring boot app. Spring Boot provides `SpringApplication` class to bootstrap a Spring application that will be started from a main() method using static SpringApplication.run method.

```
package com.websystique.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;
```

```
import com.websystique.springboot.configuration.JpaConfiguration;


@Import(JpaConfiguration.class)
@SpringBootApplication(scanBasePackages={"com.websystique.springboot"})
public class SpringBootCRUDApp {

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

This class is annotated with `@SpringBootApplication` which is actually the combination of [shortcut] `@EnableAutoConfiguration`, `@Configuration` & `@ComponentScan`. You can choose either of them.

Spring Boot @EnableAutoConfiguration attempts to automatically configure your Spring application based on the jar dependencies that you have added. Since we have added spring-boot-starter-web, Spring boot will setup the Spring configuration for a web-application.

3. JPA configuation

In this configuration class, we are doing a lot: Creating datasource [using Hikari connection pooling], creating EntityManagerFactory, setting up transaction manager, referring to Spring-data repositories etc.

- Spring Data `@EnableJpaRepositories`: @EnableJpaRepositories Annotation enables JPA repositories. It will scan the specified packages for Spring Data repositories. by default, it will look into current package for Spring-data repositories.
- Spring Boot `DataSourceProperties` : DataSourceProperties is the helper class for configuration of a data source. Interesting point is that we can map the properties right from .yml files, thanks to hierarchical data. Matching-name properties from .yml will be mapped directly to properties of DataSourceProperties object.
- Spring Boot `DataSourceBuilder` : DataSourceBuilder is a builder that can help creating a datasource using the mapped properties.
- Additionally,if a datasource property is missing in DataSourceProperties [maxPoolSize e.g.], we can still take the advantage of good old `@Value`annotation to map it form property file to actual object property.

```
package com.websystique.springboot.configuration;

import java.util.Properties;

import javax.naming.NamingException;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;

import org.apache.commons.lang3.StringUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.core.env.Environment;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement

import com.zaxxer.hikari.HikariDataSource;

@Configuration
@EnableJpaRepositories(basePackages = "com.websystique.springboot.repositorie
        entityManagerFactoryRef = "entityManagerFactory",
        transactionManagerRef = "transactionManager")
@EnableTransactionManagement
public class JpaConfiguration {

    @Autowired
    private Environment environment;

    @Value("${datasource.sampleapp.maxPoolSize:10}")
```

```
    private int maxPoolSize;

    /*
     * Populate SpringBoot DataSourceProperties object directly from applicat
     * based on prefix.Thanks to .yml, Hierachical data is mapped out of the
     * properties of DataSourceProperties object].
     */
    @Bean
    @Primary
    @ConfigurationProperties(prefix = "datasource.sampleapp")
    public DataSourceProperties dataSourceProperties(){
        return new DataSourceProperties();
    }

    /*
     * Configure HikariCP pooled DataSource.
     */
    @Bean
    public DataSource dataSource() {
        DataSourceProperties dataSourceProperties = dataSourceProperties();
            HikariDataSource dataSource = (HikariDataSource) DataSourceBuilde
                    .create(dataSourceProperties.getClassLoader())
                    .driverClassName(dataSourceProperties.getDriverClassName(
                    .url(dataSourceProperties.getUrl())
                    .username(dataSourceProperties.getUsername())
                    .password(dataSourceProperties.getPassword())
                    .type(HikariDataSource.class)
                    .build();
            dataSource.setMaximumPoolSize(maxPoolSize);
            return dataSource;
    }

    /*
     * Entity Manager Factory setup.
     */
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() thro
        LocalContainerEntityManagerFactoryBean factoryBean = new LocalContain
        factoryBean.setDataSource(dataSource());
        factoryBean.setPackagesToScan(new String[] { "com.websystique.springb
        factoryBean.setJpaVendorAdapter(jpaVendorAdapter());
        factoryBean.setJpaProperties(jpaProperties());
        return factoryBean;
    }

    /*
     * Provider specific adapter.
     */
    @Bean
    public JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new HibernateJp
        return hibernateJpaVendorAdapter;
    }

    /*
     * Here you can specify any provider specific properties.
     */
    private Properties jpaProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.getRequiredProperty("
        properties.put("hibernate.hbm2ddl.auto", environment.getRequiredPrope
        properties.put("hibernate.show_sql", environment.getRequiredProperty(
        properties.put("hibernate.format_sql", environment.getRequiredPropert
        if(StringUtils.isNotEmpty(environment.getRequiredProperty("datasource
            properties.put("hibernate.default_schema", environment.getRequire
        }
        return properties;
    }

    @Bean
    @Autowired
    public PlatformTransactionManager transactionManager(EntityManagerFactory
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(emf);
        return txManager;
    }

}
```

4. Property file [application.yml]

Although traditional .properties would just do fine, Spring Boot's SpringApplication class also supports YAML out of the box provided SnakeYAML library is on class-path which usually would be due to starters. YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data. YAML file is also known as streams, containing several documents, each separated by three dashes (—). A line beginning with "—" may be used to explicitly denote the beginning of a new YAML document.YAML specification is a good read to know more about them.

`src/main/resources/application.yml`

```
server:
  port: 8080
  contextPath: /SpringBootCRUDApp
---
spring:
  profiles: local,default
datasource:
  sampleapp:
    url: jdbc:h2:~/test
    username: SA
    password:
    driverClassName: org.h2.Driver
    defaultSchema:
    maxPoolSize: 10
    hibernate:
      hbm2ddl.method: create-drop
      show_sql: true
      format_sql: true
      dialect: org.hibernate.dialect.H2Dialect
---
spring:
  profiles: prod
datasource:
  sampleapp:
    url: jdbc:mysql://localhost:3306/websystique
    username: myuser
    password: mypassword
    driverClassName: com.mysql.jdbc.Driver
    defaultSchema:
    maxPoolSize: 20
    hibernate:
      hbm2ddl.method: update
      show_sql: true
      format_sql: true
      dialect: org.hibernate.dialect.MySQLDialect
```

- Since our app will be running on an Embedded container, we would need a way to configure the port and context-path for our app. By-default, Spring-Boot will use no context-path, and the default port would be 8080, means your application would be available at `localhost:8080`. But you can overwrite these properties by declaring them in application.yml [or application.yaml/application.properties] file. In our case, the first document [top level part, above '---' line] is the one configuring port and context path.
- Since we will be using profiles, we have created two separate documents each with it's own profile.
- By default if no profile is specified, 'default' profile is used, this is standard spring behavior. You can additionally create different profiles based on your environments and use them on run.
- In our case, we are pointing both default and local to same profile, hence letting user to run the app directly, without specifying any profile, in that case the default profile will be used. But you are free to specify a profile. While running our example [via IDE or command-line], we can provide the profile information using `-Dspring.profiles.active=local` or `-Dspring.profiles.active=prod` in VM arguments[for IDE] or on command-line `java -jar JARPATH --spring.profiles.active=local`.
- Notice the datasource part in yml file: here we are specifying all stuff related to database. Similarly if you have other aspects/concerns [security e.g.], you could create separate levels for that. We will be using `H2 database` while running under profile 'local' and `MySQL` while running with profile 'prod'.

> In case you face trouble with YAML, Online YAML editor comes handy to validate your YAML.

5. Model

`User.java`

```
package com.websystique.springboot.model;

import org.hibernate.validator.constraints.NotEmpty;
```

```
import javax.persistence.*;
import java.io.Serializable;

@Entity
@Table(name="APP_USER")
public class User implements Serializable{

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

    @NotEmpty
    @Column(name="NAME", nullable=false)
    private String name;

    @Column(name="AGE", nullable=false)
    private Integer age;

    @Column(name="SALARY", nullable=false)
    private double salary;

--- getter/setter omitted to save space
}
```

6. Spring-Data repositories

This one is rather simple.

```
package com.websystique.springboot.repositories;

import com.websystique.springboot.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    User findByName(String name);

}
```

That's it for spring-data part. Interface `JpaRepository` packs a punch. It provides all the CRUD operations by-default using id as the key. In case you need to lookup on a property other than id, you could just create a 'camelCase' signature with that property, spring-data will itself generate the implementation and execute the appropriate SQL to get the data out from database. spring-data `@Query` annotation goes a step further by allowing you to write the JPQL or even native SQL yourself instead of relying on spring-data to do that. One could as well extend from `CrudRepository` instead of JpaRepository but JpaRepository provides some goodies like paging and sorting which most of the time is needed in a FE application.

7. Service

Our controller will be using this service for all user-related operations. Service in turn uses our spring-data repository to access and update the user.

```
+ expand source
```

```
+ expand source
```

8. Controllers

We have two controllers in our application. One for handling the view and other for handling the REST API calls, coming from Our AngularJS based Front-end.

```
package com.websystique.springboot.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class AppController {
```

```
    @RequestMapping("/")
    String home(ModelMap modal) {
        modal.addAttribute("title","CRUD Example");
        return "index";
    }

    @RequestMapping("/partials/{page}")
    String partialHandler(@PathVariable("page") final String page) {
        return page;
    }

}
```

```
 + expand source
```

Additionally, a helper class to send errors [in-case any] from API in JSON format iso string.

```
package com.websystique.springboot.util;


public class CustomErrorType {

    private String errorMessage;

    public CustomErrorType(String errorMessage){
        this.errorMessage = errorMessage;
    }

    public String getErrorMessage() {
        return errorMessage;
    }

}
```

Populate MySQL database

If you look back at application.yml, we have set the hibernate **hbm2ddl** as 'create-drop' under 'local' profile, where as 'update' under 'prod' profile, just for demonstration purpose. That mean in 'local' [H2], table will be dropped and recreated at application startup so we don't need to create it manually. But in case of 'prod' [MySQL], we need to manually create the table if it does not exist. For MySQL, You can run following SQL to create table and populate dummy data.

```
create table APP_USER (
   id BIGINT NOT NULL AUTO_INCREMENT,
   name VARCHAR(30) NOT NULL,
   age  INTEGER NOT NULL,
   salary REAL NOT NULL,
   PRIMARY KEY (id)
);

/* Populate USER Table */
INSERT INTO APP_USER(name,age,salary)
VALUES ('Sam',30,70000);

INSERT INTO APP_USER(name,age,salary)
VALUES ('Tom',40,50000);

commit;
```

Front-end

Let's add a view to our MVC app. We would be using Freemarker templates in our app. Spring Boot WebMvcAutoConfiguration adds FreeMarkerViewResolver with id 'freeMarkerViewResolver' if freemarker jar is in classpath, which is the case since we are using spring-boot-starter-freemarker. It looks for resources in a loader path (externalized to `spring.freemarker.templateLoaderPath`, default 'classpath:/templates/') by surrounding the view name with a prefix and suffix (externalized to `spring.freemarker.prefix` and `spring.freemarker.suffix`, with empty and '.ftl' defaults respectively). It can be overridden by providing a bean of the same name.

Although one can develop a complete FE using freemarker itself with tons of scripts and cryptic expressions with '#' lurking around all over the page, question is should we, knowing

that we are not in 1990 anymore? I decided to use AngularJS [with ui-router] instead, using freemarker just as a container, nothing else.

Freemarker Templates

`src/main/resources/templates/index.ftl`

```
<!DOCTYPE html>

<html lang="en" ng-app="crudApp">
    <head>
        <title>${title}</title>
        <link href="css/bootstrap.css" rel="stylesheet"/>
        <link href="css/app.css" rel="stylesheet"/>
    </head>
    <body>

        <div ui-view></div>
        <script src="js/lib/angular.min.js" ></script>
        <script src="js/lib/angular-ui-router.min.js" ></script>
        <script src="js/lib/localforage.min.js" ></script>
        <script src="js/lib/ngStorage.min.js"></script>
        <script src="js/app/app.js"></script>
        <script src="js/app/UserService.js"></script>
        <script src="js/app/UserController.js"></script>
    </body>
</html>
```

`src/main/resources/templates/list.ftl`

```
+ expand source
```

Static resources

Static resources like images/css/JS in a Spring boot application are commonly located in a directory called /static (or /public or /resources or /META-INF/resources) in the classpath or from the root of the ServletContext. In this example, we are using bootstrap.css which is located in `src/main/resources/static/css`.

Error Page

By default, Spring Boot installs a 'whitelabel' error page that is shown in browser client if you encounter a server error. You can override that page, based upon the templating technology you are using. For freemarker, you can create a page with name 'error.ftl' which would be shown in case an error occurred.

`src/main/resources/templates/error.ftl`

```
<!DOCTYPE html>

<html lang="en">
<head>
    <link rel="stylesheet" type="text/css" href="css/bootstrap.css" />
</head>
<body>
    <div class="container">
        <div class="jumbotron alert-danger">
            <h1>Oops. Something went wrong</h1>
            <h2>${status} ${error}</h2>
        </div>
    </div>
</body>

</html>
```

AngularJs [ui-router based app]

`src/main/resources/static/js/app.js`

```
+ expand source
```

`src/main/resources/static/js/UserService.js`

```
+ expand source
```

src/main/resources/static/js/UserController.js

```
+ expand source
```

Run the application

Finally, Let's run the application, firstly with 'local' profile [H2]. Next shot will be with 'prod' profile [MySQL].

Via Eclipse:: Run it directly, in that case default profile will be used. In case you want a different profile to be used, create a Run configuration for you main class, specifying the profile. To do that from toolbar, select Run->Run Configurations->Arguments->VM Arguments. Add -Dspring.profiles.active=local or -Dspring.profiles.active=prod]

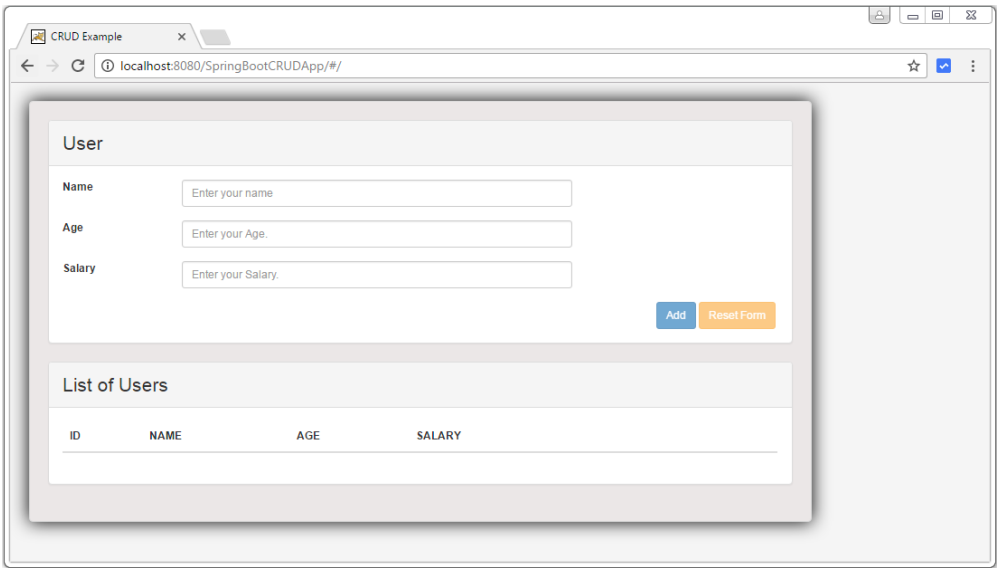
Via Command line::
On project root
$> java -jar target/SpringBootCRUDApplicationExample-1.0.0.jar –spring.profiles.active=local

Please take special note of two '-' in front of spring.profiles.active. In the blog it might be appearing as single '-' but there are in fact two '-' of them.

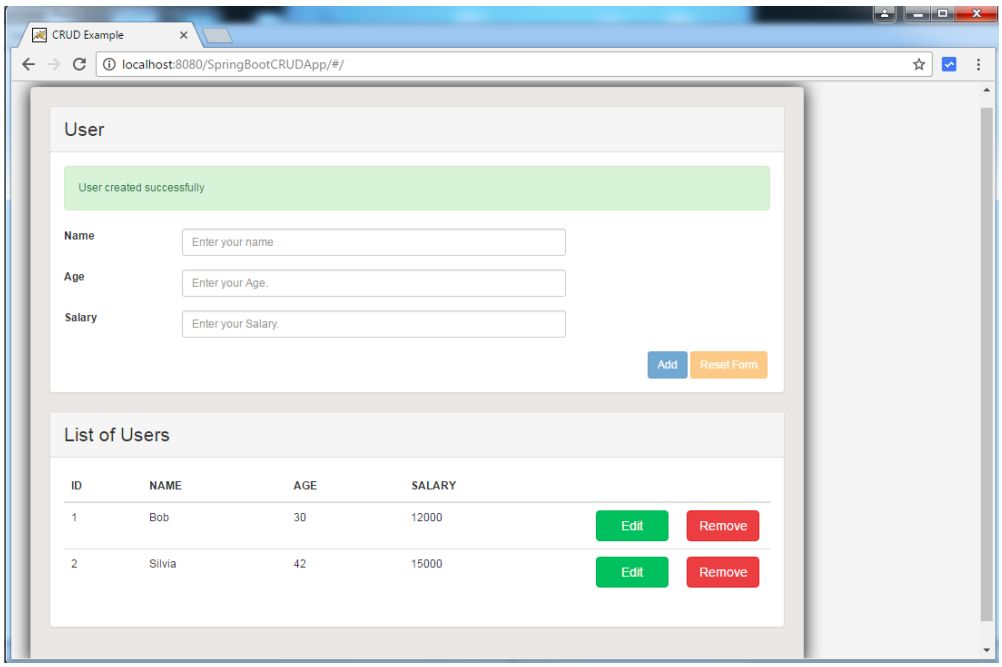
```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.3.RELEASE)

2016-12-26 17:15:35.114  INFO 4496 --- [           main] c.w.springboot.Sprin
2016-12-26 17:15:35.121  INFO 4496 --- [           main] c.w.springboot.Sprin
2016-12-26 17:15:35.225  INFO 4496 --- [           main] ationConfigEmbeddedW
2016-12-26 17:15:38.422  INFO 4496 --- [           main] trationDelegate$Bean
2016-12-26 17:15:39.458  INFO 4496 --- [           main] s.b.c.e.t.TomcatEmbe
2016-12-26 17:15:39.484  INFO 4496 --- [           main] o.apache.catalina.co
2016-12-26 17:15:39.486  INFO 4496 --- [           main] org.apache.catalina.
2016-12-26 17:15:39.692  INFO 4496 --- [ost-startStop-1] o.a.c.c.C.[.[.[/Spri
2016-12-26 17:15:39.693  INFO 4496 --- [ost-startStop-1] o.s.web.context.Cont
2016-12-26 17:15:40.031  INFO 4496 --- [ost-startStop-1] o.s.b.w.servlet.Serv
2016-12-26 17:15:40.037  INFO 4496 --- [ost-startStop-1] o.s.b.w.servlet.Filt
2016-12-26 17:15:40.038  INFO 4496 --- [ost-startStop-1] o.s.b.w.servlet.Filt
2016-12-26 17:15:40.038  INFO 4496 --- [ost-startStop-1] o.s.b.w.servlet.Filt
2016-12-26 17:15:40.038  INFO 4496 --- [ost-startStop-1] o.s.b.w.servlet.Filt
2016-12-26 17:15:40.334  INFO 4496 --- [           main] j.LocalContainerEnti
2016-12-26 17:15:40.360  INFO 4496 --- [           main] o.hibernate.jpa.inte
    name: default
    ...]
2016-12-26 17:15:40.488  INFO 4496 --- [           main] org.hibernate.Versio
2016-12-26 17:15:40.490  INFO 4496 --- [           main] org.hibernate.cfg.En
2016-12-26 17:15:40.492  INFO 4496 --- [           main] org.hibernate.cfg.En
2016-12-26 17:15:40.569  INFO 4496 --- [           main] o.hibernate.annotati
2016-12-26 17:15:40.786  INFO 4496 --- [           main] com.zaxxer.hikari.Hi
2016-12-26 17:15:41.228  INFO 4496 --- [           main] com.zaxxer.hikari.po
2016-12-26 17:15:41.311  INFO 4496 --- [           main] org.hibernate.dialec
2016-12-26 17:15:41.908  INFO 4496 --- [           main] org.hibernate.tool.h
Hibernate:
    drop table APP_USER if exists
Hibernate:
    create table APP_USER (
        id bigint generated by default as identity,
        AGE integer not null,
        NAME varchar(255) not null,
        SALARY double not null,
        primary key (id)
    )
2016-12-26 17:15:41.927  INFO 4496 --- [           main] org.hibernate.tool.h
2016-12-26 17:15:41.987  INFO 4496 --- [           main] j.LocalContainerEnti
2016-12-26 17:15:42.983  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.109  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.110  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.116  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.117  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.117  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.117  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.118  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.118  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.122  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
```

```
2016-12-26 17:15:43.123  INFO 4496 --- [           main] s.w.s.m.m.a.RequestM
2016-12-26 17:15:43.185  INFO 4496 --- [           main] o.s.w.s.handler.Simp
2016-12-26 17:15:43.185  INFO 4496 --- [           main] o.s.w.s.handler.Simp
2016-12-26 17:15:43.261  INFO 4496 --- [           main] o.s.w.s.handler.Simp
2016-12-26 17:15:43.708  INFO 4496 --- [           main] o.s.w.s.v.f.FreeMark
2016-12-26 17:15:44.002  INFO 4496 --- [           main] o.s.j.e.a.Annotation
2016-12-26 17:15:44.004  INFO 4496 --- [           main] o.s.j.e.a.Annotation
2016-12-26 17:15:44.014  INFO 4496 --- [           main] o.s.j.e.a.Annotation
2016-12-26 17:15:44.102  INFO 4496 --- [           main] s.b.c.e.t.TomcatEmbe
2016-12-26 17:15:44.110  INFO 4496 --- [           main] c.w.springboot.Sprin
```

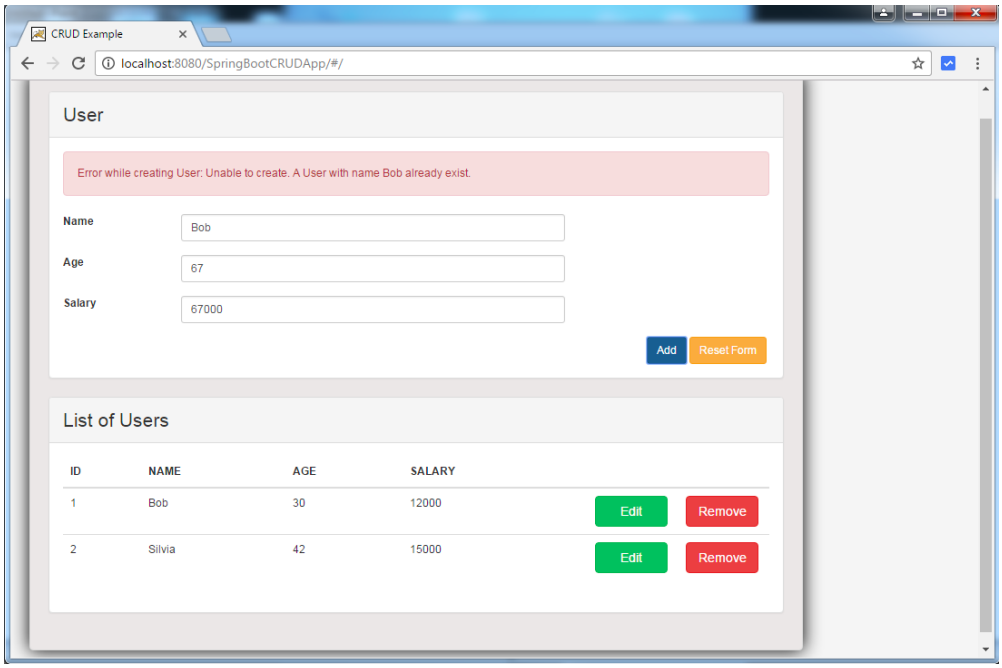
Open your browser and navigate to **http://localhost:8080/SpringBootCRUDApp/**



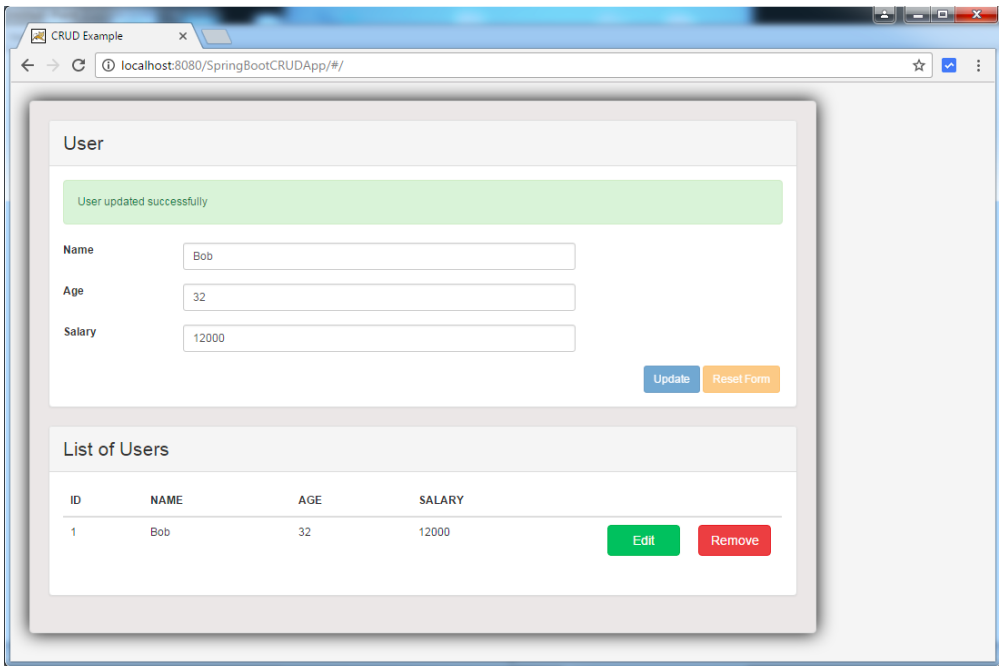
Add few users.



Try to add a user with same name as an existing user, should get an error [this is backend throwing the error, you can change the logic on backend based on your business rules].



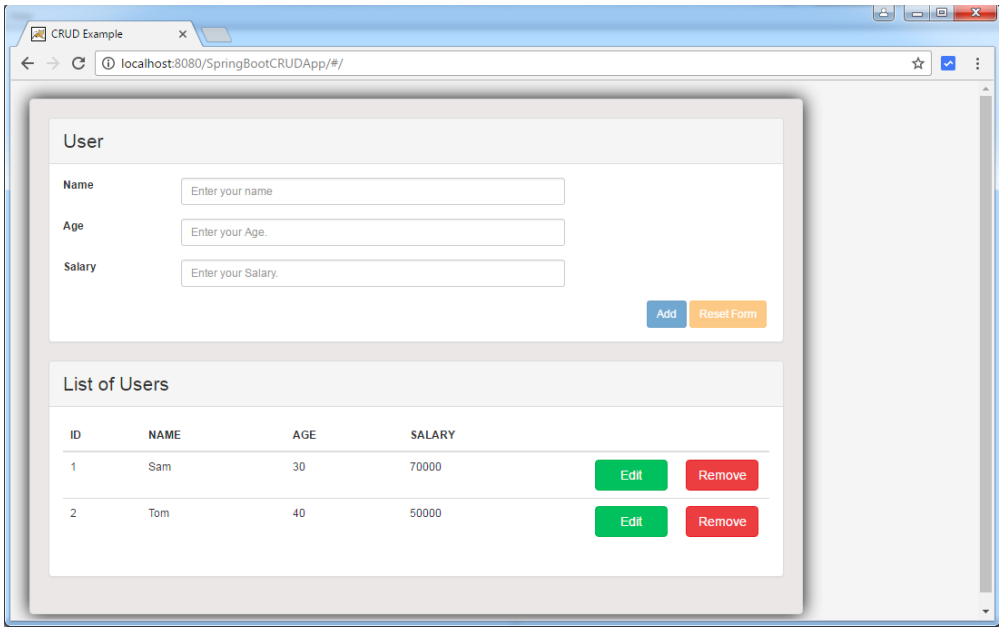
Reset the form.Remove a user, edit the other one.



Now shutdown your app and restart it using ‘prod’ profile this time.

```
+ expand source
```

Provided your mysql is up and running, you should get following, right from MySQL database this time :



Conclusion

Although the post was bit long, Spring Boot and associated concepts are fairly trivial. Spring Boot reduces the development time by many-fold, worth giving a try. The application we developed here is fully loaded and can be used in a live environment or as the base application for your own projects. Feel free to write your thoughts in comment section.

Download Source Code