# **Building web applications with Spring Boot and Kotlin**

This tutorial shows you how to build efficiently a sample blog application by combining the power of [Spring Boot](https://spring.io/projects/spring-boot/) and [Kotlin](https://kotlinlang.org/).

If you are starting with Kotlin, you can learn the language by reading the [reference documentation](https://kotlinlang.org/docs/reference/), following the online [Kotlin Koans tutorial](https://play.kotlinlang.org/) or just using [Spring Framework reference documentation](https://docs.spring.io/spring/docs/current/spring-framework-reference/) which now provides code samples in Kotlin.

Spring Kotlin support is documented in the [Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/languages.html#kotlin) and [Spring Boot](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-kotlin.html) reference documentation. If you need help, search or ask questions with the [spring and kotlin tags on StackOverflow](https://stackoverflow.com/questions/tagged/kotlin+spring) or come discuss in the #spring channel of [Kotlin Slack](https://slack.kotlinlang.org/).

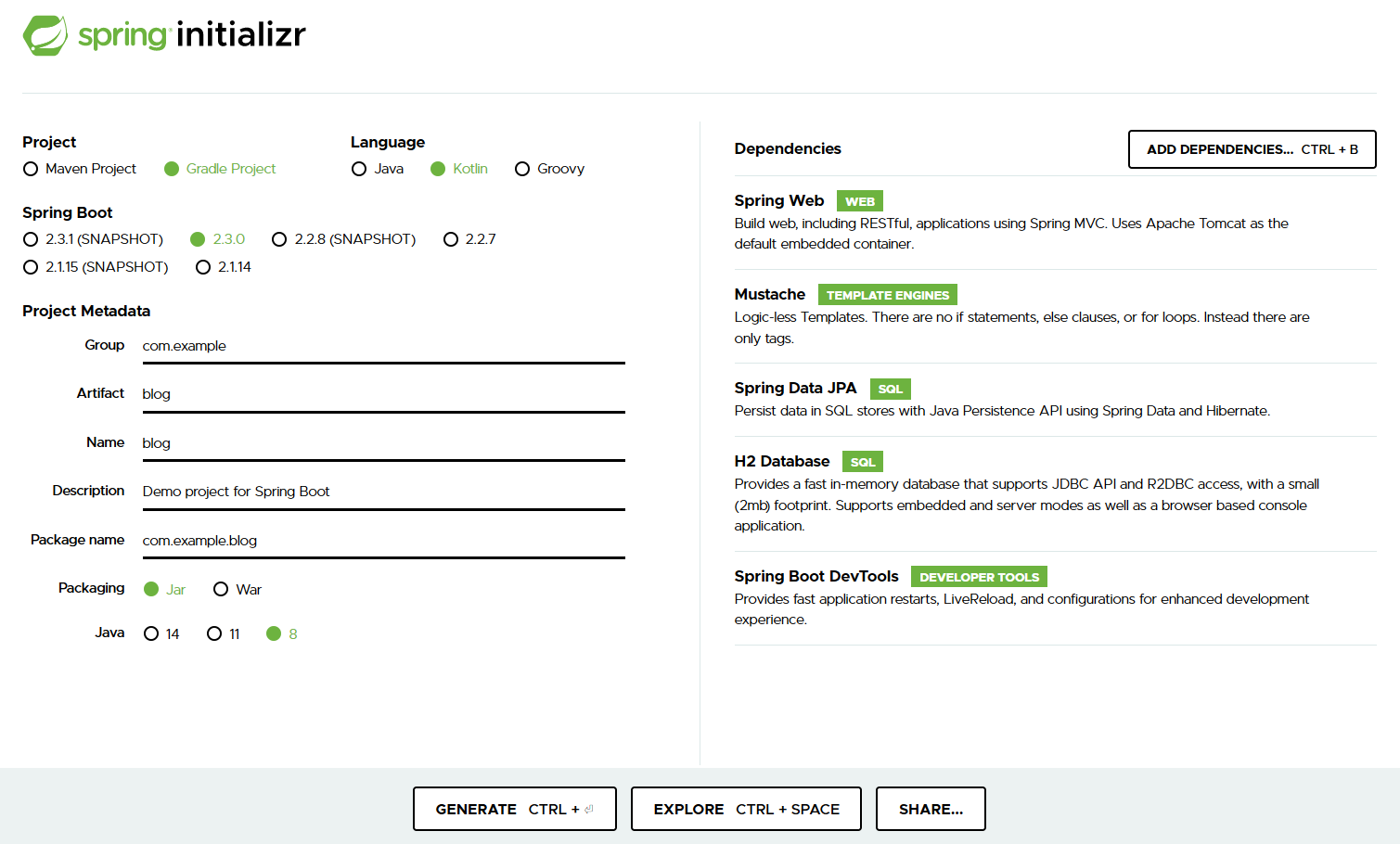
## Creating a New Project

First we need to create a Spring Boot application, which can be done in a number of ways.

### Using the Initializr Website

Visit [https://start.spring.io](https://start.spring.io/) and choose the Kotlin language. Gradle is the most commonly used build tool in Kotlin, and it provides a Kotlin DSL which is used by default when generating a Kotlin project, so this is the recommended choice. But you can also use Maven if you are more comfortable with it. Notice that you can use <https://start.spring.io/#!language=kotlin&type=gradle-project> to have Kotlin and Gradle selected by default.

1. Select "Gradle Project" or let the default "Maven Project" depending on which build tool you want to use
2. Enter the following artifact coordinates: blog
3. Add the following dependencies:
   * Spring Web
   * Mustache
   * Spring Data JPA
   * H2 Database
   * Spring Boot DevTools
4. Click "Generate Project".



The .zip file contains a standard project in the root directory, so you might want to create an empty directory before you unpack it.

### Using command line

You can use the Initializr HTTP API [from the command line](https://docs.spring.io/initializr/docs/current/reference/html/#command-line) with, for example, curl on a UN\*X like system:

$ mkdir blog && cd blog

$ curl https://start.spring.io/starter.zip -d language=kotlin -d dependencies=web,mustache,jpa,h2,devtools -d packageName=com.example.blog -d name=Blog -o blog.zip

**COPY**

Add -d type=gradle-project if you want to use Gradle.

### Using IntelliJ IDEA

Spring Initializr is also integrated in IntelliJ IDEA Ultimate edition and allows you to create and import a new project without having to leave the IDE for the command-line or the web UI.

To access the wizard, go to File | New | Project, and select Spring Initializr.

Follow the steps of the wizard to use the following parameters:

* Artifact: "blog"
* Type: Maven project or Gradle Project
* Language: Kotlin
* Name: "Blog"
* Dependencies: "Spring Web Starter", "Mustache", "Spring Data JPA", "H2 Database" and "Spring Boot DevTools"

## Gradle Build

## Maven Build

## Understanding the generated Application

src/main/kotlin/com/example/blog/BlogApplication.kt

package com.example.blog

import org.springframework.boot.autoconfigure.SpringBootApplication

import org.springframework.boot.runApplication

@SpringBootApplication

class BlogApplication

fun main(args: Array<String>) {

runApplication<BlogApplication>(\*args)

}

**COPY**

Compared to Java, you can notice the lack of semicolons, the lack of brackets on empty class (you can add some if you need to declare beans via @Bean annotation) and the use of runApplication top level function. runApplication<BlogApplication>(\*args) is Kotlin idiomatic alternative to SpringApplication.run(BlogApplication::class.java, \*args) and can be used to customize the application with following syntax.

src/main/kotlin/com/example/blog/BlogApplication.kt

fun main(args: Array<String>) {

runApplication<BlogApplication>(\*args) {

setBannerMode(Banner.Mode.OFF)

}

}

**COPY**

## Writing your first Kotlin controller

Let’s create a simple controller to display a simple web page.

src/main/kotlin/com/example/blog/HtmlController.kt

package com.example.blog

import org.springframework.stereotype.Controller

import org.springframework.ui.Model

import org.springframework.ui.set

import org.springframework.web.bind.annotation.GetMapping

@Controller

class HtmlController {

@GetMapping("/")

fun blog(model: Model): String {

model["title"] = "Blog"

return "blog"

}

}

**COPY**

Notice that we are using here a [Kotlin extension](https://kotlinlang.org/docs/reference/extensions.html) that allows to add Kotlin functions or operators to existing Spring types. Here we import the org.springframework.ui.set extension function in order to be able to write model["title"] = "Blog" instead of model.addAttribute("title", "Blog"). The [Spring Framework KDoc API](https://docs.spring.io/spring-framework/docs/current/kdoc-api/spring-framework/) lists all the Kotlin extensions provided to enrich the Java API.

We also need to create the associated Mustache templates.

src/main/resources/templates/header.mustache

<html>

<head>

<title>{{title}}</title>

</head>

<body>

**COPY**

src/main/resources/templates/footer.mustache

</body>

</html>

**COPY**

src/main/resources/templates/blog.mustache

{{> header}}

<h1>{{title}}</h1>

{{> footer}}

**COPY**

Start the web application by running the main function of BlogApplication.kt, and go to <http://localhost:8080/>, you should see a sober web page with a "Blog" headline.

## Testing with JUnit 5

JUnit 5 now used by default in Spring Boot provides various features very handy with Kotlin, including [autowiring of constructor/method parameters](https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testcontext-junit-jupiter-di) which allows to use non-nullable val properties and the possibility to use @BeforeAll/@AfterAll on regular non-static methods.

### Writing JUnit 5 tests in Kotlin

For the sake of this example, let’s create an integration test in order to demonstrate various features:

* We use real sentences between backticks instead of camel-case to provide expressive test function names
* JUnit 5 allows to inject constructor and method parameters, which is a good fit with Kotlin read-only and non-nullable properties
* This code leverages getForObject and getForEntity Kotlin extensions (you need to import them)

src/test/kotlin/com/example/blog/IntegrationTests.kt

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

class IntegrationTests(@Autowired val restTemplate: TestRestTemplate) {

@Test

fun `Assert blog page title, content and status code`() {

val entity = restTemplate.getForEntity<String>("/")

assertThat(entity.statusCode).isEqualTo(HttpStatus.OK)

assertThat(entity.body).contains("<h1>Blog</h1>")

}

}

**COPY**

### Test instance lifecycle

Sometimes you need to execute a method before or after all tests of a given class. Like Junit 4, JUnit 5 requires by default these methods to be static (which translates to [companion object](https://kotlinlang.org/docs/reference/object-declarations.html#companion-objects) in Kotlin, which is quite verbose and not straightforward) because test classes are instantiated one time per test.

But Junit 5 allows you to change this default behavior and instantiate test classes one time per class. This can be done in [various ways](https://junit.org/junit5/docs/current/user-guide/#writing-tests-test-instance-lifecycle), here we will use a property file to change the default behavior for the whole project:

src/test/resources/junit-platform.properties

junit.jupiter.testinstance.lifecycle.default = per\_class

**COPY**

With this configuration, we can now use @BeforeAll and @AfterAll annotations on regular methods like shown in updated version of IntegrationTests above.

src/test/kotlin/com/example/blog/IntegrationTests.kt

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

class IntegrationTests(@Autowired val restTemplate: TestRestTemplate) {

@BeforeAll

fun setup() {

println(">> Setup")

}

@Test

fun `Assert blog page title, content and status code`() {

println(">> Assert blog page title, content and status code")

val entity = restTemplate.getForEntity<String>("/")

assertThat(entity.statusCode).isEqualTo(HttpStatus.OK)

assertThat(entity.body).contains("<h1>Blog</h1>")

}

@Test

fun `Assert article page title, content and status code`() {

println(">> TODO")

}

@AfterAll

fun teardown() {

println(">> Tear down")

}

}

**COPY**

## Creating your own extensions

Instead of using util classes with abstract methods like in Java, it is usual in Kotlin to provide such functionalities via Kotlin extensions. Here we are going to add a format() function to the existing LocalDateTime type in order to generate text with the english date format.

src/main/kotlin/com/example/blog/Extensions.kt

fun LocalDateTime.format() = this.format(englishDateFormatter)

private val daysLookup = (1..31).associate { it.toLong() to getOrdinal(it) }

private val englishDateFormatter = DateTimeFormatterBuilder()

.appendPattern("yyyy-MM-dd")

.appendLiteral(" ")

.appendText(ChronoField.DAY\_OF\_MONTH, daysLookup)

.appendLiteral(" ")

.appendPattern("yyyy")

.toFormatter(Locale.ENGLISH)

private fun getOrdinal(n: Int) = when {

n in 11..13 -> "${n}th"

n % 10 == 1 -> "${n}st"

n % 10 == 2 -> "${n}nd"

n % 10 == 3 -> "${n}rd"

else -> "${n}th"

}

fun String.toSlug() = toLowerCase()

.replace("\n", " ")

.replace("[^a-z\\d\\s]".toRegex(), " ")

.split(" ")

.joinToString("-")

.replace("-+".toRegex(), "-")

**COPY**

We will leverage these extensions in the next section.

## Persistence with JPA

In order to make lazy fetching working as expected, entities should be open as described in [KT-28525](https://youtrack.jetbrains.com/issue/KT-28525). We are going to use the Kotlin allopen plugin for that purpose.

With Gradle:

build.gradle.kts

plugins {

...

kotlin("plugin.allopen") version "1.4.32"

}

allOpen {

annotation("javax.persistence.Entity")

annotation("javax.persistence.Embeddable")

annotation("javax.persistence.MappedSuperclass")

}

**COPY**

Or with Maven:

pom.xml

<plugin>

<artifactId>kotlin-maven-plugin</artifactId>

<groupId>org.jetbrains.kotlin</groupId>

<configuration>

...

<compilerPlugins>

...

<plugin>all-open</plugin>

</compilerPlugins>

<pluginOptions>

<option>all-open:annotation=javax.persistence.Entity</option>

<option>all-open:annotation=javax.persistence.Embeddable</option>

<option>all-open:annotation=javax.persistence.MappedSuperclass</option>

</pluginOptions>

</configuration>

</plugin>

**COPY**

Then we create our model by using Kotlin [primary constructor concise syntax](https://kotlinlang.org/docs/reference/classes.html#constructors) which allows to declare at the same time the properties and the constructor parameters.

src/main/kotlin/com/example/blog/Entities.kt

@Entity

class Article(

var title: String,

var headline: String,

var content: String,

@ManyToOne var author: User,

var slug: String = title.toSlug(),

var addedAt: LocalDateTime = LocalDateTime.now(),

@Id @GeneratedValue var id: Long? = null)

@Entity

class User(

var login: String,

var firstname: String,

var lastname: String,

var description: String? = null,

@Id @GeneratedValue var id: Long? = null)

**COPY**

Notice that we are using here our String.toSlug() extension to provide a default argument to the slug parameter of Article constructor. Optional parameters with default values are defined at the last position in order to make it possible to omit them when using positional arguments (Kotlin also supports [named arguments](https://kotlinlang.org/docs/reference/functions.html#named-arguments)). Notice that in Kotlin it is not unusual to group concise class declarations in the same file.

|  |  |
| --- | --- |
|  | Here we don’t use [data classes](https://kotlinlang.org/docs/reference/data-classes.html) with val properties because JPA is not designed to work with immutable classes or the methods generated automatically by data classes. If you are using other Spring Data flavor, most of them are designed to support such constructs so you should use classes like data class User(val login: String, …​) when using Spring Data MongoDB, Spring Data JDBC, etc. |

|  |  |
| --- | --- |
|  | While Spring Data JPA makes it possible to use natural IDs (it could have been the login property in User class) via [Persistable](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.entity-persistence.saving-entites), it is not a good fit with Kotlin due to [KT-6653](https://youtrack.jetbrains.com/issue/KT-6653), that’s why it is recommended to always use entities with generated IDs in Kotlin. |

We also declare our Spring Data JPA repositories as following.

src/main/kotlin/com/example/blog/Repositories.kt

interface ArticleRepository : CrudRepository<Article, Long> {

fun findBySlug(slug: String): Article?

fun findAllByOrderByAddedAtDesc(): Iterable<Article>

}

interface UserRepository : CrudRepository<User, Long> {

fun findByLogin(login: String): User?

}

**COPY**

And we write JPA tests to check whether basic use cases work as expected.

src/test/kotlin/com/example/blog/RepositoriesTests.kt

@DataJpaTest

class RepositoriesTests @Autowired constructor(

val entityManager: TestEntityManager,

val userRepository: UserRepository,

val articleRepository: ArticleRepository) {

@Test

fun `When findByIdOrNull then return Article`() {

val juergen = User("springjuergen", "Juergen", "Hoeller")

entityManager.persist(juergen)

val article = Article("Spring Framework 5.0 goes GA", "Dear Spring community ...", "Lorem ipsum", juergen)

entityManager.persist(article)

entityManager.flush()

val found = articleRepository.findByIdOrNull(article.id!!)

assertThat(found).isEqualTo(article)

}

@Test

fun `When findByLogin then return User`() {

val juergen = User("springjuergen", "Juergen", "Hoeller")

entityManager.persist(juergen)

entityManager.flush()

val user = userRepository.findByLogin(juergen.login)

assertThat(user).isEqualTo(juergen)

}

}

**COPY**

|  |  |
| --- | --- |
|  | We use here the CrudRepository.findByIdOrNull Kotlin extension provided by default with Spring Data, which is a nullable variant of the Optional based CrudRepository.findById. Read the great [Null is your friend, not a mistake](https://medium.com/@elizarov/null-is-your-friend-not-a-mistake-b63ff1751dd5) blog post for more details. |

## Implementing the blog engine

We update the "blog" Mustache templates.

src/main/resources/templates/blog.mustache

{{> header}}

<h1>{{title}}</h1>

<div class="articles">

{{#articles}}

<section>

<header class="article-header">

<h2 class="article-title"><a href="/article/{{slug}}">{{title}}</a></h2>

<div class="article-meta">By <strong>{{author.firstname}}</strong>, on <strong>{{addedAt}}</strong></div>

</header>

<div class="article-description">

{{headline}}

</div>

</section>

{{/articles}}

</div>

{{> footer}}

**COPY**

And we create an "article" new one.

src/main/resources/templates/article.mustache

{{> header}}

<section class="article">

<header class="article-header">

<h1 class="article-title">{{article.title}}</h1>

<p class="article-meta">By <strong>{{article.author.firstname}}</strong>, on <strong>{{article.addedAt}}</strong></p>

</header>

<div class="article-description">

{{article.headline}}

{{article.content}}

</div>

</section>

{{> footer}}

**COPY**

We update the HtmlController in order to render blog and article pages with the formatted date. ArticleRepository and MarkdownConverter constructor parameters will be automatically autowired since HtmlController has a single constructor (implicit @Autowired).

src/main/kotlin/com/example/blog/HtmlController.kt

@Controller

class HtmlController(private val repository: ArticleRepository) {

@GetMapping("/")

fun blog(model: Model): String {

model["title"] = "Blog"

model["articles"] = repository.findAllByOrderByAddedAtDesc().map { it.render() }

return "blog"

}

@GetMapping("/article/{slug}")

fun article(@PathVariable slug: String, model: Model): String {

val article = repository

.findBySlug(slug)

?.render()

?: throw ResponseStatusException(HttpStatus.NOT\_FOUND, "This article does not exist")

model["title"] = article.title

model["article"] = article

return "article"

}

fun Article.render() = RenderedArticle(

slug,

title,

headline,

content,

author,

addedAt.format()

)

data class RenderedArticle(

val slug: String,

val title: String,

val headline: String,

val content: String,

val author: User,

val addedAt: String)

}

**COPY**

Then, we add data initialization to a new BlogConfiguration class.

src/main/kotlin/com/example/blog/BlogConfiguration.kt

@Configuration

class BlogConfiguration {

@Bean

fun databaseInitializer(userRepository: UserRepository,

articleRepository: ArticleRepository) = ApplicationRunner {

val smaldini = userRepository.save(User("smaldini", "Stéphane", "Maldini"))

articleRepository.save(Article(

title = "Reactor Bismuth is out",

headline = "Lorem ipsum",

content = "dolor sit amet",

author = smaldini

))

articleRepository.save(Article(

title = "Reactor Aluminium has landed",

headline = "Lorem ipsum",

content = "dolor sit amet",

author = smaldini

))

}

}

**COPY**

|  |  |
| --- | --- |
|  | Notice the usage of named parameters to make the code more readable. |

And we also update the integration tests accordingly.

src/test/kotlin/com/example/blog/IntegrationTests.kt

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

class IntegrationTests(@Autowired val restTemplate: TestRestTemplate) {

@BeforeAll

fun setup() {

println(">> Setup")

}

@Test

fun `Assert blog page title, content and status code`() {

println(">> Assert blog page title, content and status code")

val entity = restTemplate.getForEntity<String>("/")

assertThat(entity.statusCode).isEqualTo(HttpStatus.OK)

assertThat(entity.body).contains("<h1>Blog</h1>", "Reactor")

}

@Test

fun `Assert article page title, content and status code`() {

println(">> Assert article page title, content and status code")

val title = "Reactor Aluminium has landed"

val entity = restTemplate.getForEntity<String>("/article/${title.toSlug()}")

assertThat(entity.statusCode).isEqualTo(HttpStatus.OK)

assertThat(entity.body).contains(title, "Lorem ipsum", "dolor sit amet")

}

@AfterAll

fun teardown() {

println(">> Tear down")

}

}

**COPY**

Start (or restart) the web application, and go to <http://localhost:8080/>, you should see the list of articles with clickable links to see a specific article.

## Exposing HTTP API

We are now going to implement the HTTP API via @RestController annotated controllers.

src/main/kotlin/com/example/blog/HttpControllers.kt

@RestController

@RequestMapping("/api/article")

class ArticleController(private val repository: ArticleRepository) {

@GetMapping("/")

fun findAll() = repository.findAllByOrderByAddedAtDesc()

@GetMapping("/{slug}")

fun findOne(@PathVariable slug: String) =

repository.findBySlug(slug) ?: throw ResponseStatusException(HttpStatus.NOT\_FOUND, "This article does not exist")

}

@RestController

@RequestMapping("/api/user")

class UserController(private val repository: UserRepository) {

@GetMapping("/")

fun findAll() = repository.findAll()

@GetMapping("/{login}")

fun findOne(@PathVariable login: String) =

repository.findByLogin(login) ?: throw ResponseStatusException(HttpStatus.NOT\_FOUND, "This user does not exist")

}

**COPY**

For tests, instead of integration tests, we are going to leverage @WebMvcTest and [Mockk](https://mockk.io/) which is similar to [Mockito](https://site.mockito.org/) but better suited for Kotlin.

Since @MockBean and @SpyBean annotations are specific to Mockito, we are going to leverage [SpringMockK](https://github.com/Ninja-Squad/springmockk) which provides similar @MockkBean and @SpykBean annotations for Mockk.

With Gradle:

build.gradle.kts

testImplementation("org.springframework.boot:spring-boot-starter-test") {

exclude(module = "junit")

exclude(module = "mockito-core")

}

testImplementation("org.junit.jupiter:junit-jupiter-api")

testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine")

testImplementation("com.ninja-squad:springmockk:3.0.1")

**COPY**

Or with Maven:

pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</exclusion>

<exclusion>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.ninja-squad</groupId>

<artifactId>springmockk</artifactId>

<version>3.0.1</version>

<scope>test</scope>

</dependency>

**COPY**

src/test/kotlin/com/example/blog/HttpControllersTests.kt

@WebMvcTest

class HttpControllersTests(@Autowired val mockMvc: MockMvc) {

@MockkBean

private lateinit var userRepository: UserRepository

@MockkBean

private lateinit var articleRepository: ArticleRepository

@Test

fun `List articles`() {

val juergen = User("springjuergen", "Juergen", "Hoeller")

val spring5Article = Article("Spring Framework 5.0 goes GA", "Dear Spring community ...", "Lorem ipsum", juergen)

val spring43Article = Article("Spring Framework 4.3 goes GA", "Dear Spring community ...", "Lorem ipsum", juergen)

every { articleRepository.findAllByOrderByAddedAtDesc() } returns listOf(spring5Article, spring43Article)

mockMvc.perform(get("/api/article/").accept(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk)

.andExpect(content().contentType(MediaType.APPLICATION\_JSON))

.andExpect(jsonPath("\$.[0].author.login").value(juergen.login))

.andExpect(jsonPath("\$.[0].slug").value(spring5Article.slug))

.andExpect(jsonPath("\$.[1].author.login").value(juergen.login))

.andExpect(jsonPath("\$.[1].slug").value(spring43Article.slug))

}

@Test

fun `List users`() {

val juergen = User("springjuergen", "Juergen", "Hoeller")

val smaldini = User("smaldini", "Stéphane", "Maldini")

every { userRepository.findAll() } returns listOf(juergen, smaldini)

mockMvc.perform(get("/api/user/").accept(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk)

.andExpect(content().contentType(MediaType.APPLICATION\_JSON))

.andExpect(jsonPath("\$.[0].login").value(juergen.login))

.andExpect(jsonPath("\$.[1].login").value(smaldini.login))

}

}

**COPY**

|  |  |
| --- | --- |
|  | $ needs to be escaped in strings as it is used for string interpolation. |

## Configuration properties

In Kotlin, the recommended way to manage your application properties is to leverage @ConfigurationProperties with @ConstructorBinding in order to be able to use read-only properties.

src/main/kotlin/com/example/blog/BlogProperties.kt

@ConstructorBinding

@ConfigurationProperties("blog")

data class BlogProperties(var title: String, val banner: Banner) {

data class Banner(val title: String? = null, val content: String)

}

**COPY**

Then we enable it at BlogApplication level.

src/main/kotlin/com/example/blog/BlogApplication.kt

@SpringBootApplication

@EnableConfigurationProperties(BlogProperties::class)

class BlogApplication {

// ...

}

**COPY**

To generate [your own metadata](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-annotation-processor) in order to get these custom properties recognized by your IDE, [kapt should be configured](https://kotlinlang.org/docs/reference/kapt.html) with the spring-boot-configuration-processor dependency as following.

build.gradle.kts

plugins {

...

kotlin("kapt") version "1.4.32"

}

dependencies {

...

kapt("org.springframework.boot:spring-boot-configuration-processor")

}

**COPY**

|  |  |
| --- | --- |
|  | Note that some features (such as detecting the default value or deprecated items) are not working due to limitations in the model kapt provides. Also annotation processing is not yet supported with Maven due to [KT-18022](https://youtrack.jetbrains.com/issue/KT-18022), see [initializr#438](https://github.com/spring-io/initializr/issues/438) for more details. |

In IntelliJ IDEA:

* Make sure Spring Boot plugin in enabled in menu File | Settings | Plugins | Spring Boot
* Enable annotation processing via menu File | Settings | Build, Execution, Deployment | Compiler | Annotation Processors | Enable annotation processing
* Since [Kapt is not yet integrated in IDEA](https://youtrack.jetbrains.com/issue/KT-15040), you need to run manually the command ./gradlew kaptKotlin to generate the metadata

Your custom properties should now be recognized when editing application.properties (autocomplete, validation, etc.).

src/main/resources/application.properties

blog.title=Blog

blog.banner.title=Warning

blog.banner.content=The blog will be down tomorrow.

**COPY**

Edit the template and the controller accordingly.

src/main/resources/templates/blog.mustache

{{> header}}

<div class="articles">

{{#banner.title}}

<section>

<header class="banner">

<h2 class="banner-title">{{banner.title}}</h2>

</header>

<div class="banner-content">

{{banner.content}}

</div>

</section>

{{/banner.title}}

...

</div>

{{> footer}}

**COPY**

src/main/kotlin/com/example/blog/HtmlController.kt

@Controller

class HtmlController(private val repository: ArticleRepository,

private val properties: BlogProperties) {

@GetMapping("/")

fun blog(model: Model): String {

model["title"] = properties.title

model["banner"] = properties.banner

model["articles"] = repository.findAllByOrderByAddedAtDesc().map { it.render() }

return "blog"

}

// ...

**COPY**

Restart the web application, refresh <http://localhost:8080/>, you should see the banner on the blog homepage.

## Conclusion

We have now finished to build this sample Kotlin blog application. The source code [is available on Github](https://github.com/spring-guides/tut-spring-boot-kotlin). You can also have a look to [Spring Framework](https://docs.spring.io/spring/docs/current/spring-framework-reference/languages.html#kotlin) and [Spring Boot](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-kotlin.html) reference documentation if you need more details on specific features.