endava

Secure. Test. Repeat

A guide to Spring Security integration testing





99 little bugs in the code,

99 little bugs.

Track one down, patch it around,

There's 113 little bugs in the code.





endava

YouTube

youtube.com/@laurspilca

@laurspilca

Laurențiu Spilcă

Spring Security IN ACTION

SECOND EDITION

Laurențiu Spilcă
Foreword by Joe Grandja



MANNING

TION Second Edition
by Joe Grandja

nuch easier to secure
s. This powerful frame-
s end to end, with "secure
e features that help you
authentication and pro-
And like everything else
n source, and backed by

n updates this bestselling
eep coverage of OAuth2/
uration using the new
explanations and rele-
d your own authoriza-
s, and prevent cross-site

orization

s
ions

opers.

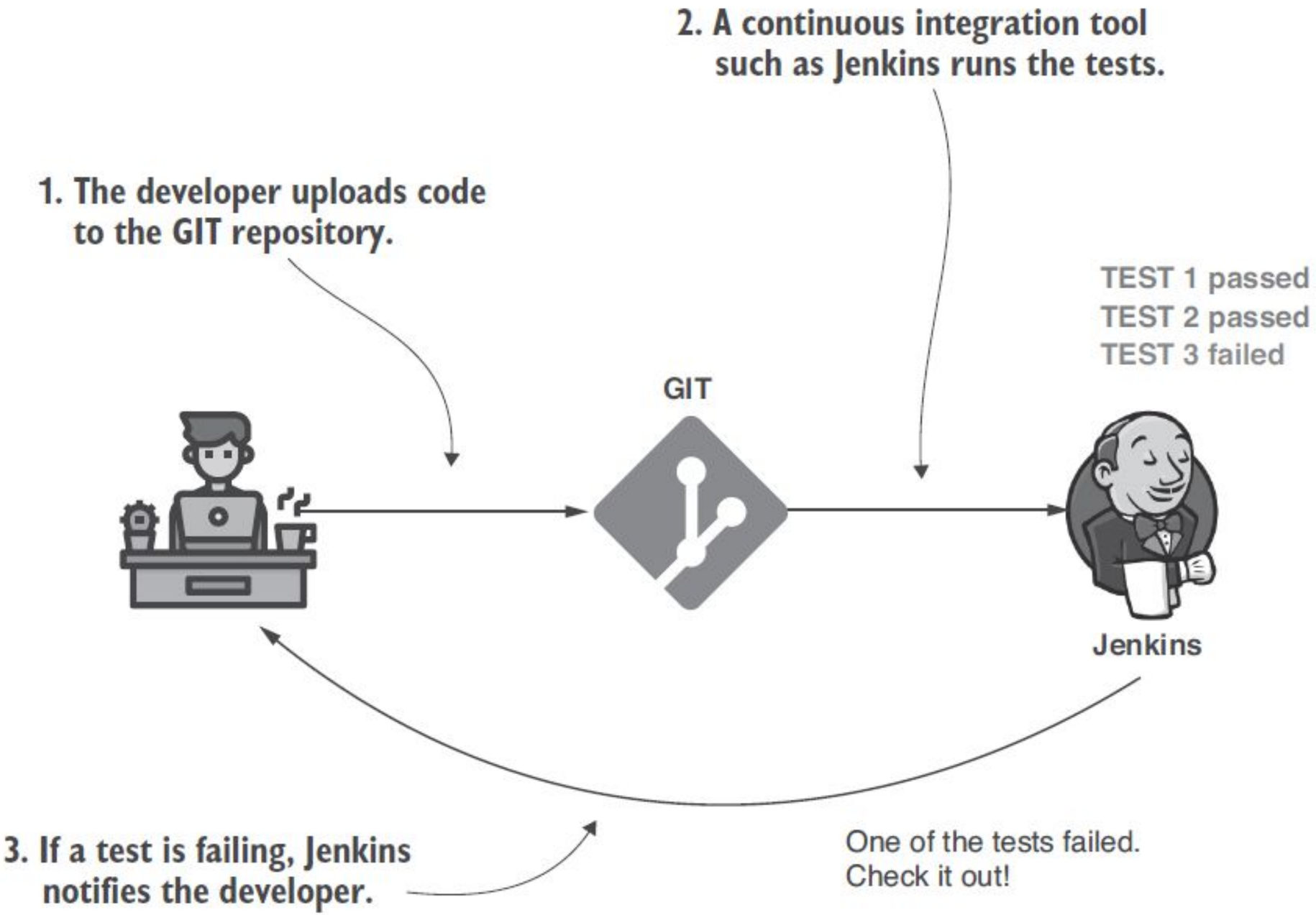
pring developer and an

"This go-to guide demystifies security and provides actionable code using the industry-leading Spring Security framework."
—Josh Long
Spring Developer Advocate

"Easy to grasp theory backed up by usable code."
—Ana-Maria Mihalceanu
Developer Advocate

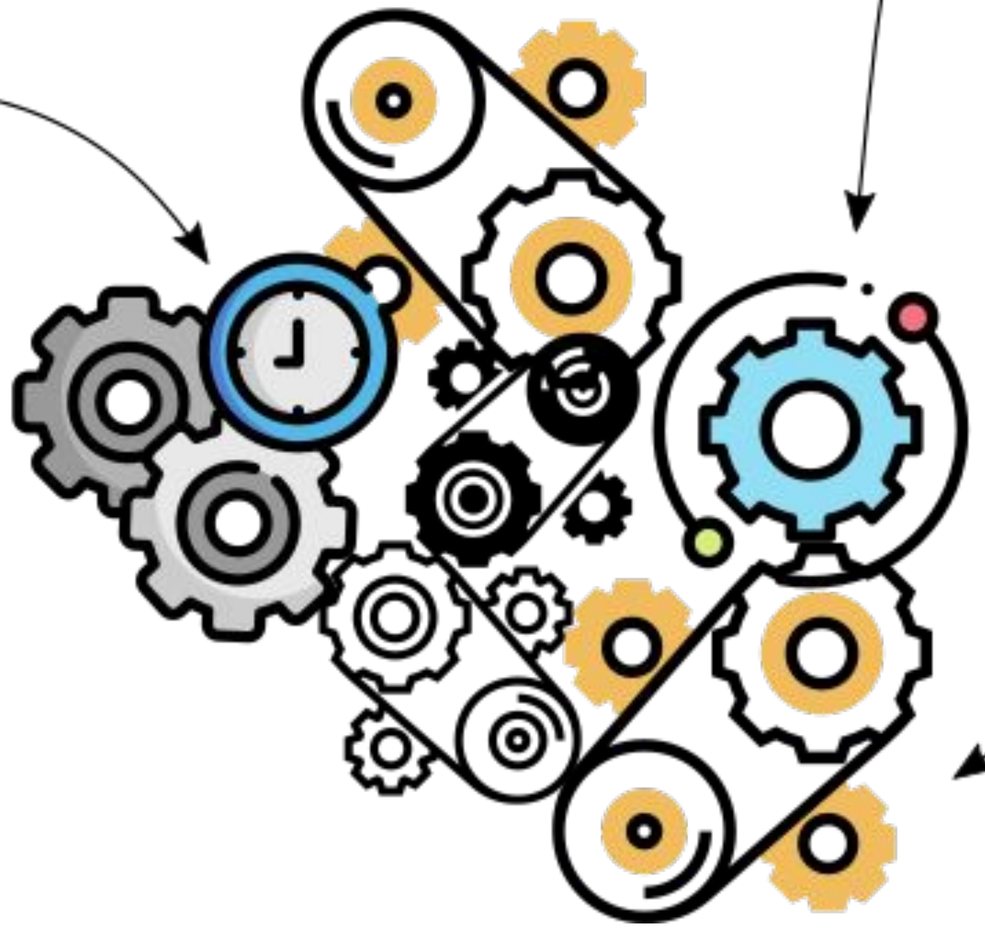
"A welcome follow-on to the excellent first edition, incorporating numerous updates to Spring Security. Check out this book!"
—Mark Heckler, Author of *Spring Boot: Up & Running!*

"Laurențiu makes the complex simple—perfect for a book about security. A must-read!"
—Thomas Vitale, Author of *Cloud Native Spring in Action*

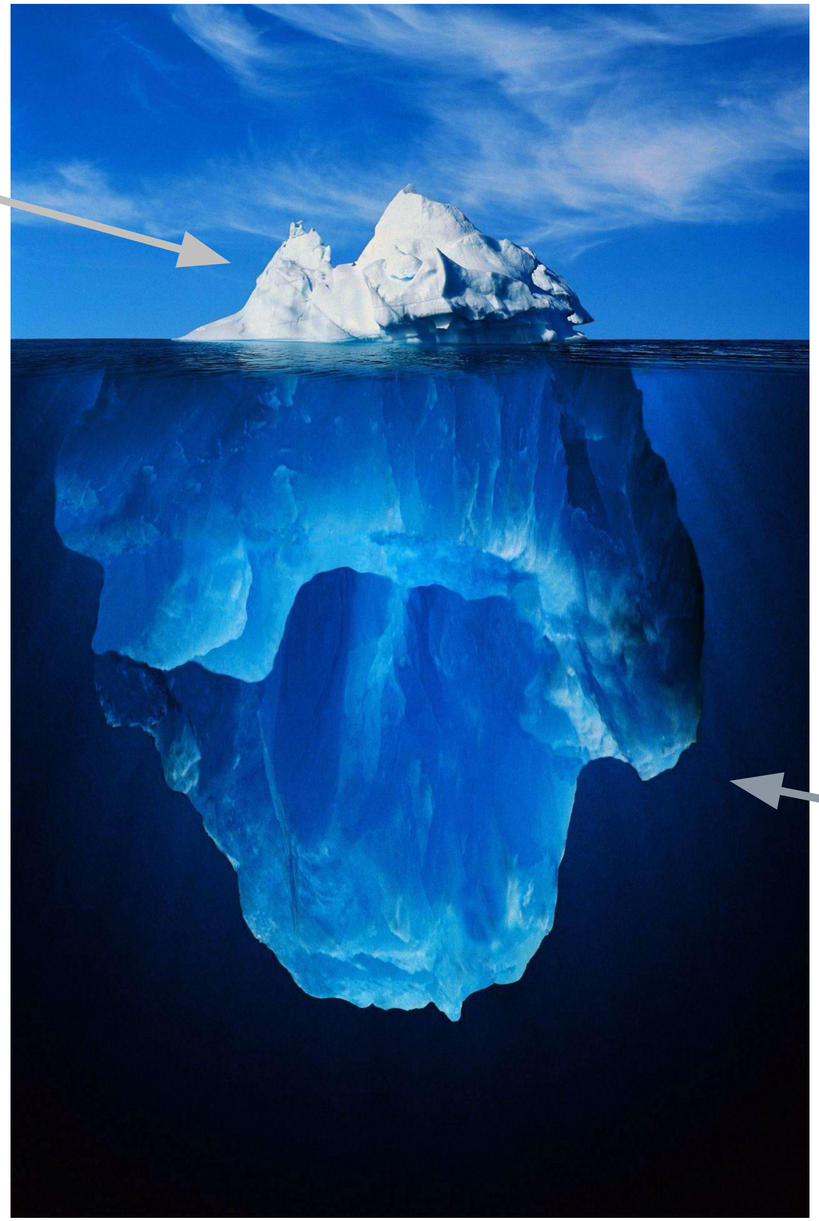

2. A continuous integration tool such as Jenkins runs the tests.
1. The developer uploads code to the GIT repository.
TEST 1 passed
TEST 2 passed
TEST 3 failed
GIT
Jenkins
3. If a test is failing, Jenkins notifies the developer.
One of the tests failed.
Check it out!

Dependencies

Your app



Your code



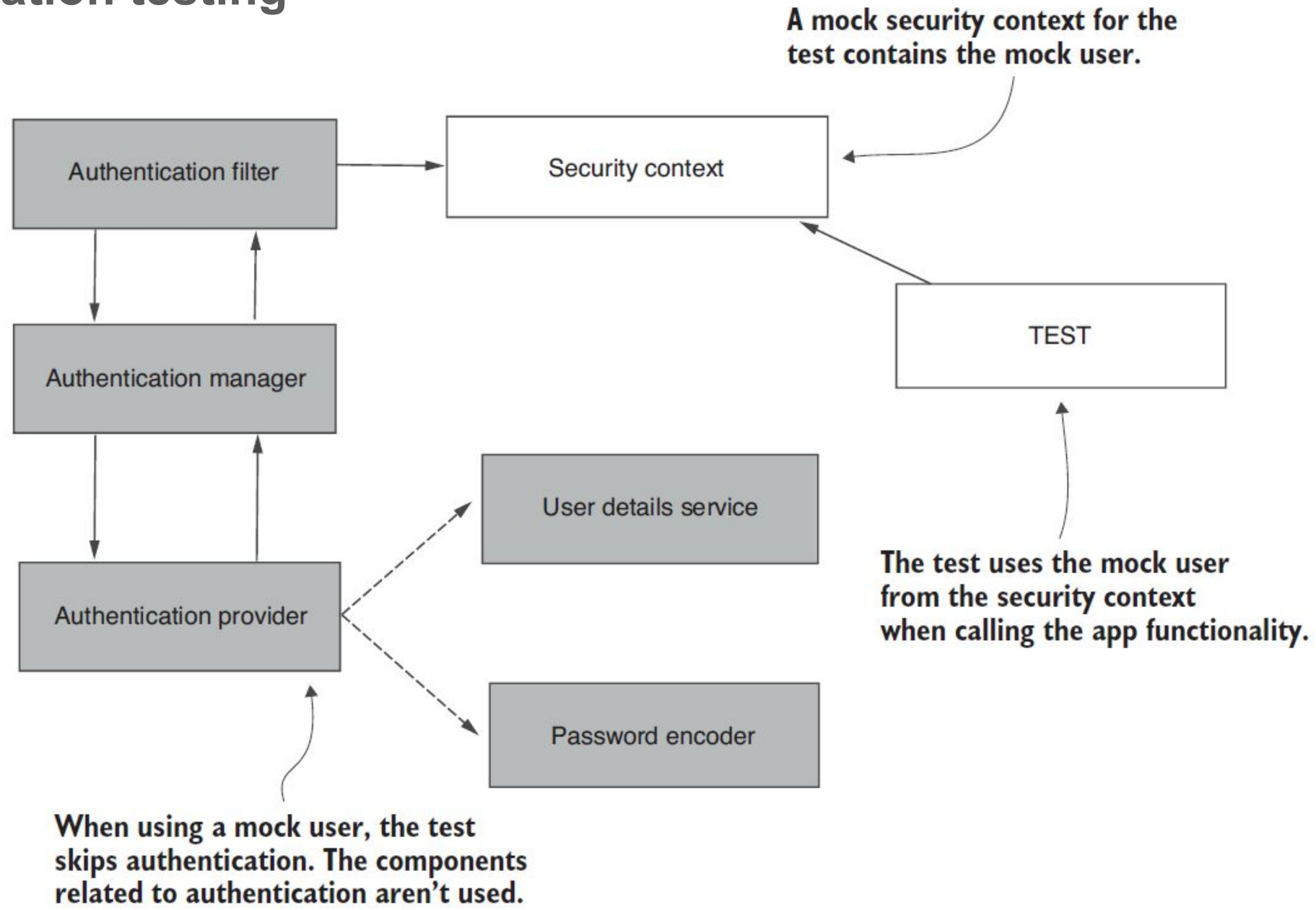
Dependencies

0. What do you test?

1. Testing authorization using mock users
2. Testing authorization using user management
3. Testing authorization with custom authentication

Make your tests meaningful!

Authorization testing



1. Mock Users

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

  @Autowired
  private MockMvc mvc;

  @Test
  public void helloUnauthenticated() throws Exception {
    mvc.perform(get("/hello"))
         .andExpect(status().isUnauthorized());
  }

}
```

When performing a GET request for the / hello path, we expect to get back a response with the status Unauthorized.

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

  // Omitted code

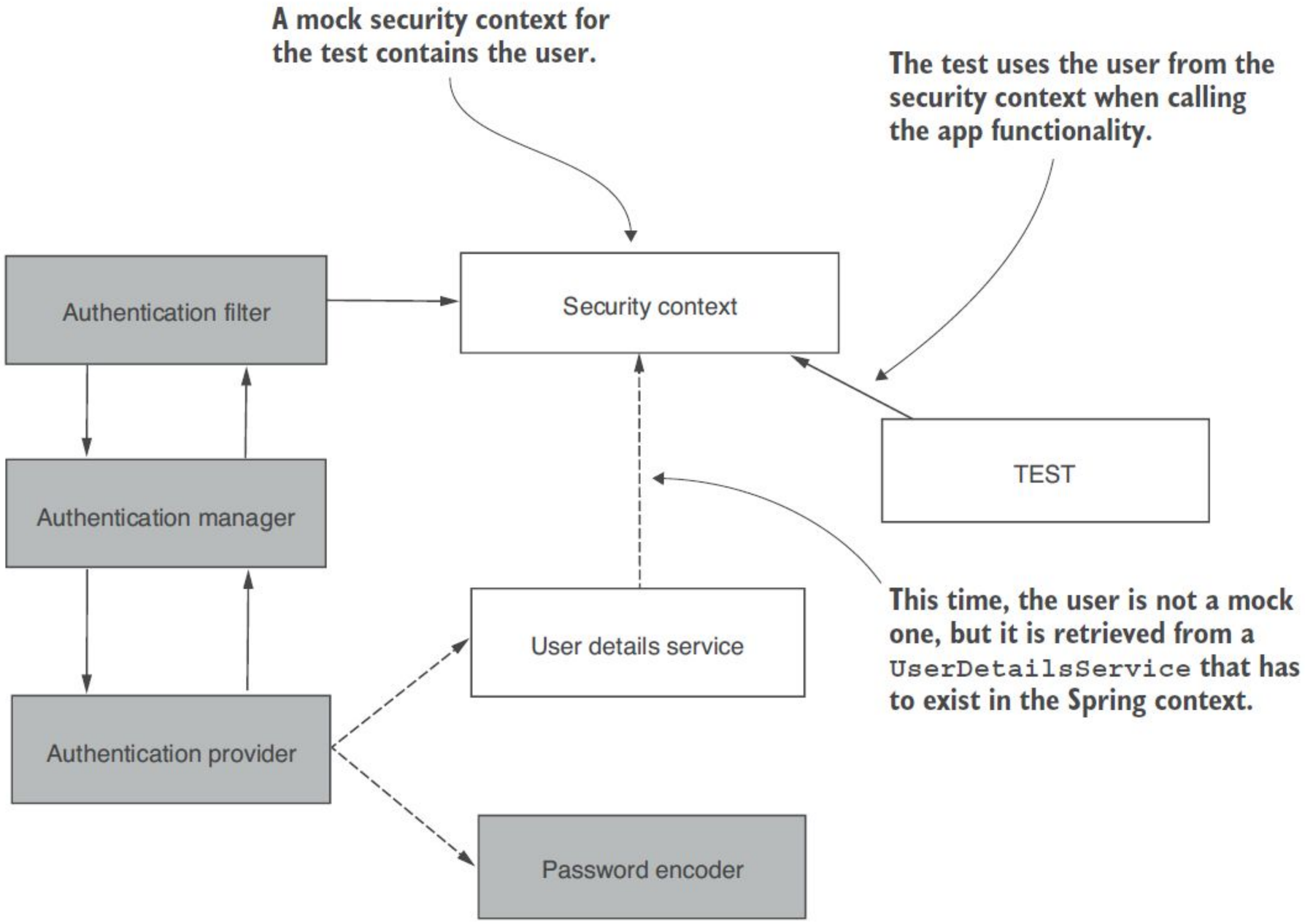
  @Test
  @WithMockUser(username = "mary")
  public void helloAuthenticated() throws Exception {
    mvc.perform(get("/hello"))
         .andExpect(content().string("Hello, mary!"))
         .andExpect(status().isOk());
    }
}
```

Sets up a username for the mock user

DEMO



2. Managed Users


A mock security context for the test contains the user.
The test uses the user from the security context when calling the app functionality.
Authentication filter
Security context
Authentication manager
TEST
User details service
This time, the user is not a mock one, but it is retrieved from a UserDetailsService that has to exist in the Spring context.
Authentication provider
Password encoder

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

  @Autowired
  private MockMvc mvc;

  @Test
  @WithUserDetails("john")
  public void helloAuthenticated() throws Exception {
    mvc.perform(get("/hello"))
        .andExpect(status().isOk());
  }

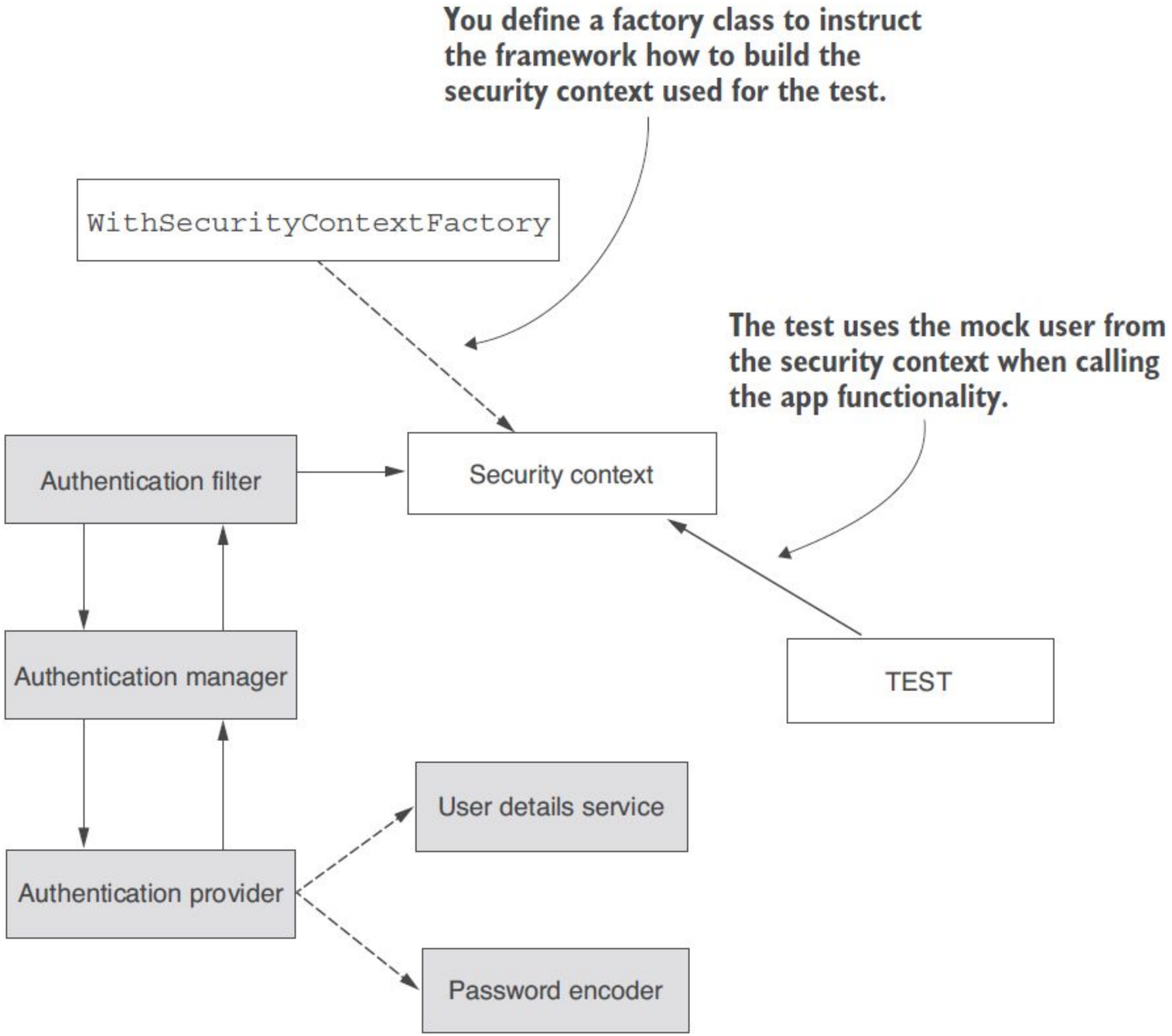
}
```

Loads the user John using the UserDetailsService for running the test scenario

DEMO



3. Custom Authentication


You define a factory class to instruct the framework how to build the security context used for the test.
WithSecurityContextFactory
The test uses the mock user from the security context when calling the app functionality.
Authentication filter
Security context
Authentication manager
TEST
User details service
Authentication provider
Password encoder

Step 1

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
public @interface WithCustomUser {

  String username();
}
```

We define a custom annotation to use for the test methods for which we need to customize the test security context.

Step 2

```
WithSecurityContextFactory


public class CustomSecurityContextFactory
  implements WithSecurityContextFactory<WithCustomUser> {

  @Override
  public SecurityContext createSecurityContext() {
    ...
  }
}
```

We implement a factory class in which we build the custom test security context. The framework uses this factory class to build the security context when it finds a test method annotated with the custom annotation we created in step 1.

Step 3

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext
  (factory = CustomSecurityContextFactory.class)
public @interface WithCustomUser {

  String username();
}
```

We link the custom annotation created in step 1 to the factory class created in step 2. This way, the framework knows which factory class to use to create the test security context for a test method annotated with out custom annotation.

DEMO




VOXXED DAYS
CLUJ NAPOCA
15 - 16 OCTOBER, 2024
CFP OPEN
https://vdcluj24.cfp.dev/

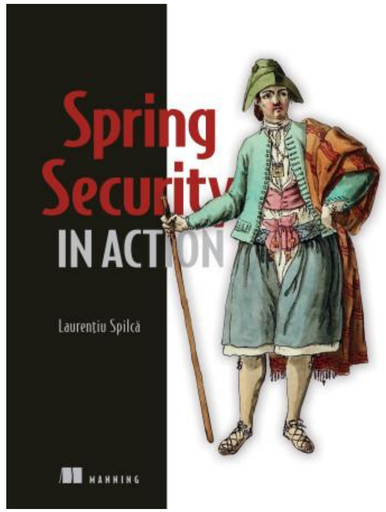




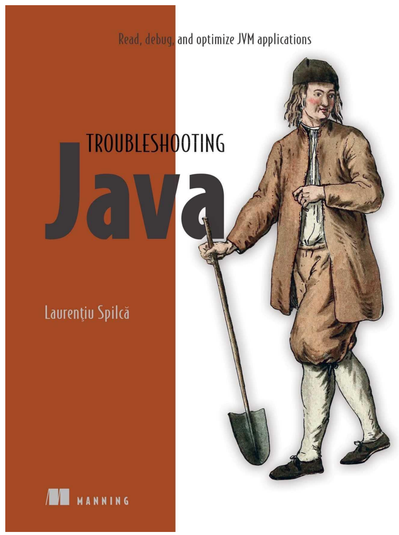


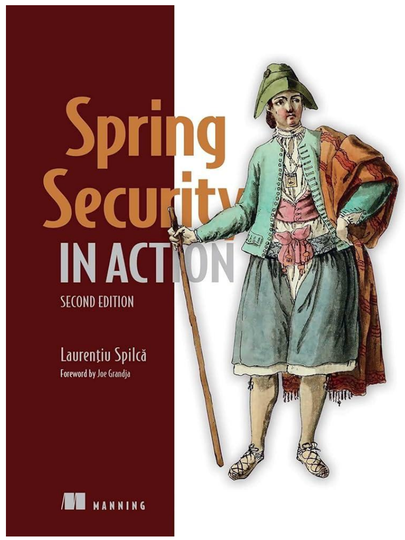


endava









YouTube

youtube.com/@laurspilca

@laurspilca

Laurentiu Spilca

