White Box Testing and Coverage

Structural testing, often referred to as white-box testing, is a rigorous methodology for evaluating the internal workings of a software application. This technique delves into the application's source code, architecture, and design, offering a detailed view of its internal pathways. Unlike black-box testing, which assesses the software's external functionality without regard to its internal mechanisms, structural testing demands an intimate understanding of the codebase. This approach allows testers to meticulously examine execution paths, logic flows, and the outcomes of various code segments.

The Crucial Role of Structural Testing

1. **Enhanced Code Coverage**: One of the paramount benefits of structural testing is its ability to achieve exhaustive code coverage. It meticulously examines branches, loops, and individual statements within the code, ensuring that no part is left untested. This level of scrutiny is vital for uncovering errors that might elude functional testing, thereby bolstering the software's reliability and performance.
2. **Detection of Concealed Bugs**: Structural testing shines in its ability to unearth bugs that lurk beneath the surface, invisible to more superficial testing methodologies. It probes into the software's inner mechanisms, identifying edge cases and unique conditions that might otherwise remain undiscovered until after deployment.
3. **Streamlined Debugging Process**: The intimate association between structural testing and the application's codebase significantly eases the debugging process. When a test fails, developers can quickly pinpoint the exact location of the fault within the code, facilitating a more efficient and targeted debugging effort.
4. **Elevated Software Quality**: By verifying that the code behaves as intended across a wide array of scenarios and inputs, structural testing contributes significantly to enhancing the software's overall quality. This comprehensive examination helps ensure a superior user experience and can substantially reduce the need for future maintenance and bug fixes.
5. **Support for Refactoring and Integration**: Structural testing is invaluable during code refactoring and the integration of new features. It provides a safety net that ensures modifications do not introduce new errors, allowing the software to evolve while maintaining its integrity and functionality.
6. **Bolstering Security**: Given its thorough exploration of code pathways, structural testing plays a key role in identifying potential security vulnerabilities. This is crucial for applications dealing with sensitive information or those that are integral to business operations, as it helps prevent exploitations that could lead to data breaches or other security incidents.

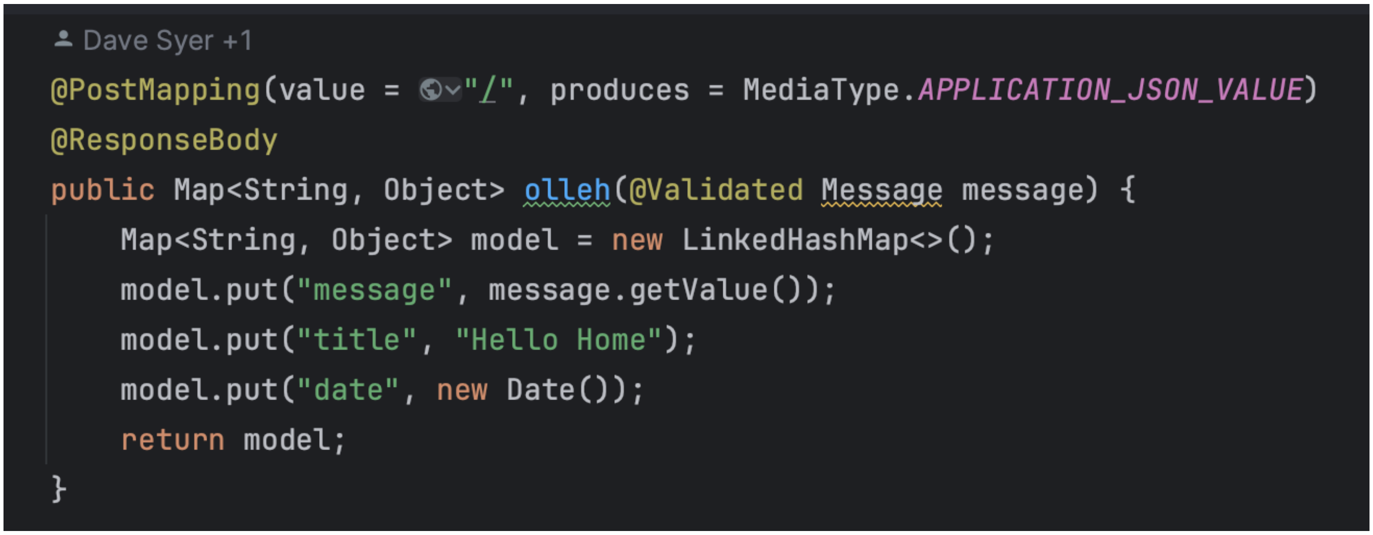
7. **Compliance with Regulatory Standards**: In certain industries, achieving comprehensive code coverage through structural testing is not just beneficial but mandated. This ensures that software adheres to rigorous quality and safety standards, an essential requirement in sectors where software reliability is paramount.

> To sum up, structural testing is indispensable for ensuring that software is not only functional but also robust, secure, and of high quality. It complements functional testing by offering a deep dive into the software's internal mechanics, thereby playing a critical role in the development of reliable, efficient, and secure applications.

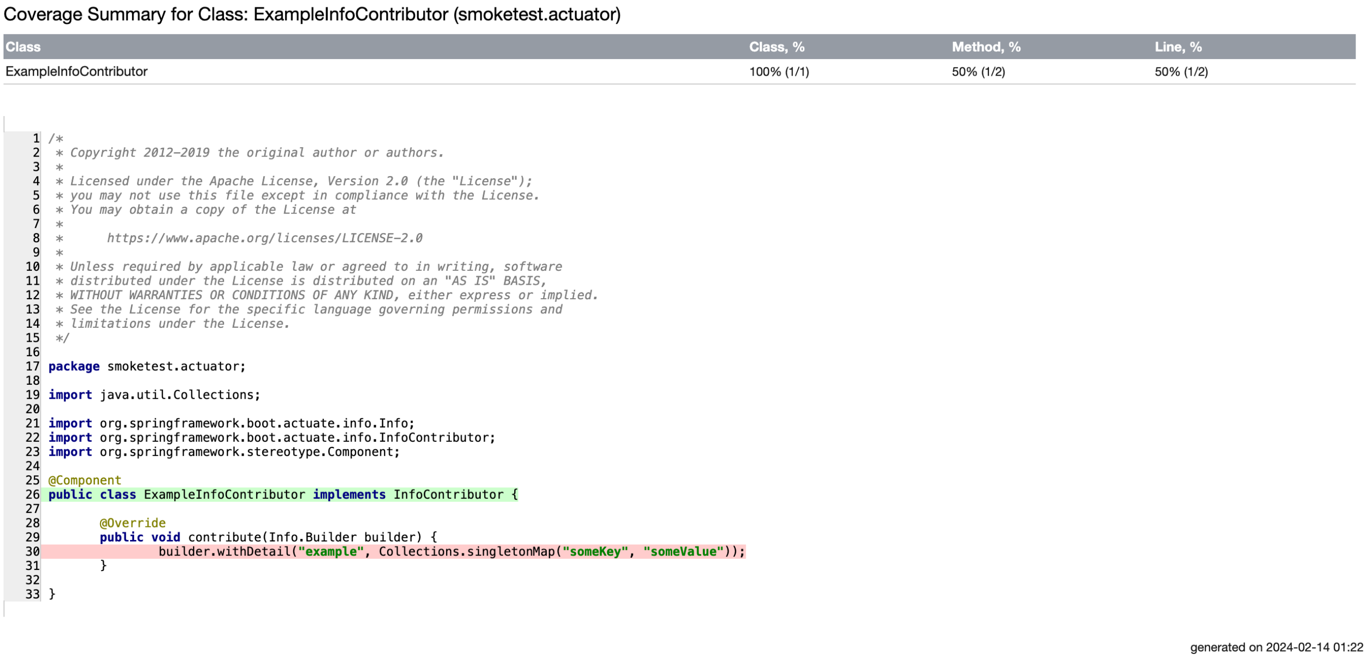
Implementation

Our test case mainly focuses on method coverage,but also improves some line coverage and branch coverage rate.

With the help of `IDEA` coverage tool, we quickly found some targets. They are:



```java
@PostMapping(value = "/", produces = MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public Map<String, Object> olleh(@Validated Message message) {
    Map<String, Object> model = new LinkedHashMap<>();
    model.put("message", message.getValue());
    model.put("title", "Hello Home");
    model.put("date", new Date());
    return model;
}
```



Coverage Summary for Class: ExampleInfoContributor (smoketest.actuator)

Class	Class, %	Method, %	Line, %
ExampleInfoContributor	100% (1/1)	50% (1/2)	50% (1/2)

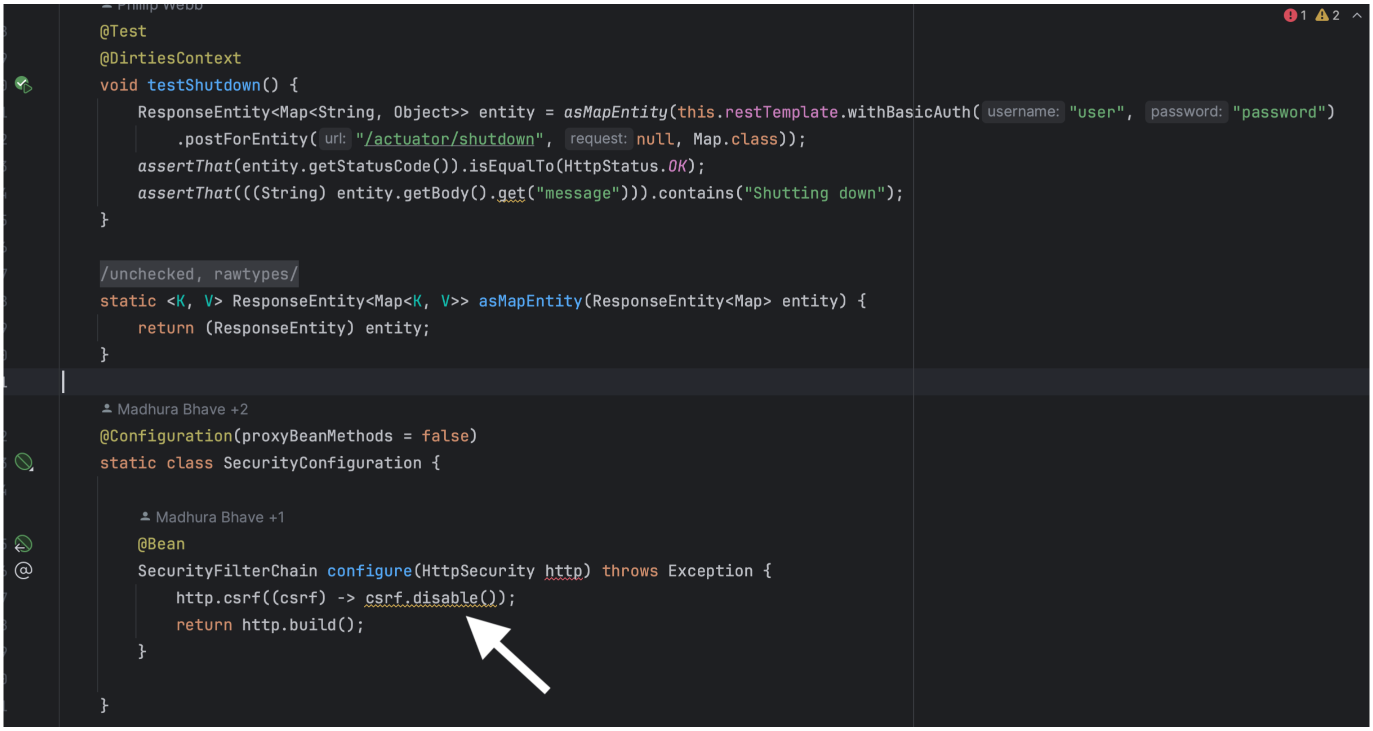
```
/*
 * Copyright 2012-2019 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package smoketest.actuator;

import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

	@Override
	public void contribute(Info.Builder builder) {
		builder.withDetail("example", Collections.singletonMap("someKey", "someValue"));
	}

}
```

generated on 2024-02-14 01:22

Both method are relatively simple, the first one handles HTTP POST requests to the root URL ("/") and return some kinds of map info such as current date etc., and the second one is mapping a KV pair into the end of actuator `info` .

Challenges Encountered

Our initial attempts to cover the first target with test cases resulted in multiple failures. Specifically, we encountered a `401 Unauthorized` status instead of the expected `200 OK` . Upon reviewing the debug logs, we discovered that Spring Security's CSRF (Cross-site request forgery) protection was blocking the POST requests.



```
Philip Webb
@Test
@DirtiesContext
void testShutdown() {
    ResponseEntity<Map<String, Object>> entity = asMapEntity(this.restTemplate.withBasicAuth( username: "user",  password: "password")
        .postForEntity( url: "/actuator/shutdown",  request: null, Map.class));
    assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(((String) entity.getBody().get("message"))).contains("Shutting down");
}

/unchecked, rawtypes/
static <K, V> ResponseEntity<Map<K, V>> asMapEntity(ResponseEntity<Map> entity) {
    return (ResponseEntity) entity;
}

Madhura Bhave +2
@Configuration(proxyBeanMethods = false)
static class SecurityConfiguration {

    Madhura Bhave +1
    @Bean
    SecurityFilterChain configure(HttpSecurity http) throws Exception {
        http.csrf((csrf) -> csrf.disable());
        return http.build();
    }

}
```

Despite disabling CSRF protection, our tests still returned a `404` status, indicating further complications possibly related to requiring a valid token or additional configuration adjustments.

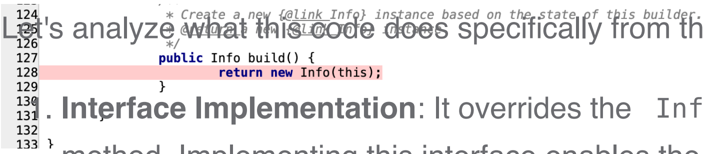
Given the complexity of Spring Security and our limited experience with it, we decided to redirect our efforts towards other methods for coverage improvement. Spring Security's comprehensive nature suggested that a deeper dive into its configuration and security mechanisms would be necessary for successful method coverage in areas protected by security policies.

We turn our attention to the second method. It seems like this code is brief, but in fact it does a lot of things under the hood. You may notice that the method itself introduce a new `Builder` class. See the below picture to take a look at this class itself.

Coverage Summary for Class: Info (org.springframework.boot.actuate.info)

Class	Method, %	Branch, %	Line, %
Info	0% (0/7)	0% (0/10)	0% (0/16)
Info$Builder	0% (0/4)	0% (0/7)	
Total	0% (0/11)	0% (0/10)	0% (0/23)

```java
/*
 * Copyright 2012-2022 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot.actuate.info;

import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Map;

import com.fasterxml.jackson.annotation.JsonAnyGetter;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonInclude.Include;

/**
 * Carries information of the application.
 * <p>
 * Each detail element can be singular or a hierarchical object such as a POJO or a nested
 * Map.
 *
 * @author Meang Akira Tanaka
 * @author Stephane Nicoll
 * @since 1.4.0
 */
@JsonInclude(Include.NON_EMPTY)
public final class Info {

       private final Map<String, Object> details;

       private Info(Builder builder) {
               Map<String, Object> content = new LinkedHashMap<>(builder.content);
               this.details = Collections.unmodifiableMap(content);
       }

       /**
        * Return the content.
        * @return the details of the info or an empty map.
        */
       @JsonAnyGetter
       public Map<String, Object> getDetails() {
               return this.details;
       }

       public Object get(String id) {
               return this.details.get(id);
       }

       @SuppressWarnings("unchecked")
       public <T> T get(String id, Class<T> type) {
               Object value = get(id);
               if (value != null && type != null && !type.isInstance(value)) {
                       throw new IllegalStateException("Info entry is not of required type [" + type.getName() + "]: " + value);
               }
               return (T) value;
       }

       @Override
       public boolean equals(Object obj) {
               if (obj == this) {
                       return true;
               }
               if (obj instanceof Info other) {
                       return this.details.equals(other.details);
               }
               return false;
       }

       @Override
       public int hashCode() {
               return this.details.hashCode();
       }

       @Override
       public String toString() {
               return getDetails().toString();
       }

       /**
        * Builder for creating immutable {@link Info} instances.
        */
       public static class Builder {

               private final Map<String, Object> content;

               public Builder() {
                       this.content = new LinkedHashMap<>();
               }

               /**
                * Record detail using given {@code key} and {@code value}.
                * @param key the detail key
                * @param value the detail value
                * @return this {@link Builder} instance
                */
               public Builder withDetail(String key, Object value) {
                       this.content.put(key, value);
                       return this;
               }

               /**
                * Record several details.
                * @param details the details
                * @return this {@link Builder} instance
                * @see #withDetail(String, Object)
                */
               public Builder withDetails(Map<String, Object> details) {
                       this.content.putAll(details);
                       return this;
               }

               /**
```



```
     * Create a new {@link Info} instance based on the state of this builder.
     * @return a new {@link Info} instance
     */
    public Info build() {
        return new Info(this);
    }

}

}
```

Let's analyze what this code does specifically from the source code perspective.

1. **Interface Implementation**: It overrides the `InfoContributor` interface's `contribute` method. Implementing this interface enables the addition of custom information to the Actuator's /info endpoint through the introduction of a `Builder` class.
2. **Detail Addition**: Utilizes the `Builder` class's withDetail method to add a key-value pair (example key with a map object { `someKey, someValue` }) as custom information.
3. **HashMap Initialization and Construction**: The Builder initializes and constructs a new HashMap, incorporating existing info content. This step ensures that any previously available information is retained and integrated with the new details.
4. **Info Map Construction**: It calls the withDetail method again to incorporate the custom information into the new HashMap, effectively building the enriched info map.
5. **Info Map Retrieval**: Upon request to the /info endpoint, the method returns the newly constructed info map, now containing both existing and newly added details.

Test case

The test case aims to verify the functionality of the override `contribute` method within the Spring Boot Actuator's `info` endpoint. It focuses on ensuring that:

- The HTTP connection is successfully established.
- The info endpoint is correctly configured and returns the expected build key.
- The contribute method functions as intended, allowing for custom information to be added to the endpoint.

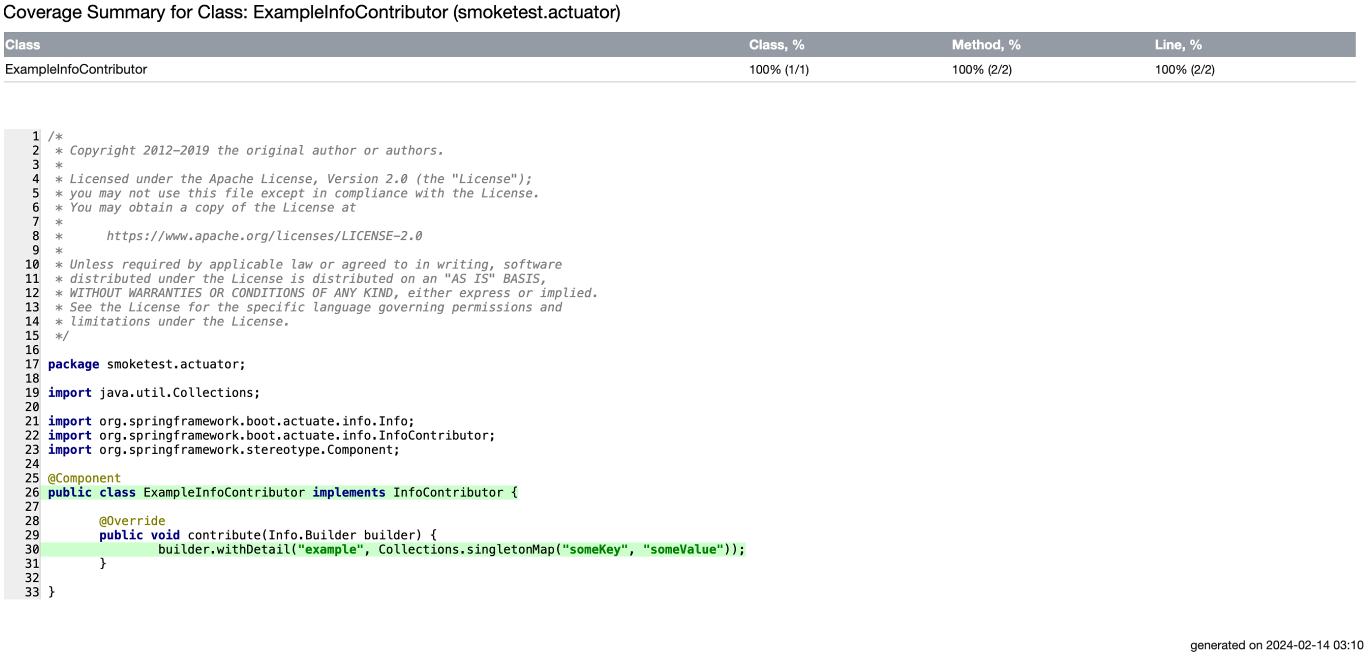
```
@SuppressWarnings("unchecked")
void testInfo() {
        ResponseEntity<Map<String, Object>> entity = asMapEntity(
                        this.restTemplate.withBasicAuth("user", "password").getForEntity
        assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(entity.getBody()).containsKey("build");
        Map<String, Object> body = entity.getBody();
        Map<String, Object> example = (Map<String, Object>) body.get("example");
        assertThat(example).containsEntry("someKey", "someValue");
}
```

Assertions

1. **HTTP Status Check**: Validates that the HTTP connection to the `info` endpoint is established, indicated by an `HttpStatus.OK` response.

2. `build` **Key Existence**: Confirms that the info endpoint is functioning correctly by checking for the presence of the `build` key, a critical component of the endpoint's response.
3. `contribute` **Method Functionality**: Verifies that the `contribute` method properly adds custom information (**"someKey": "someValue"**) to the endpoint's response.

These are the coverage output after this tested case added. You can clearly see the changes in coverage methods, lines and branches.



Coverage Summary for Class: ExampleInfoContributor (smoketest.actuator)

Class	Class, %	Method, %	Line, %
ExampleInfoContributor	100% (1/1)	100% (2/2)	100% (2/2)

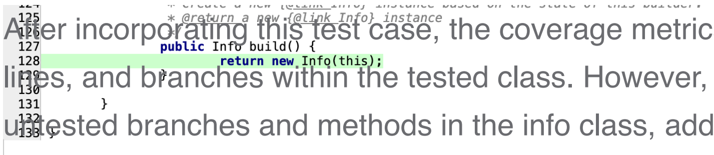
```java
/*
 * Copyright 2012-2019 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package smoketest.actuator;

import java.util.Collections;

import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

	@Override
	public void contribute(Info.Builder builder) {
		builder.withDetail("example", Collections.singletonMap("someKey", "someValue"));
	}

}
```

generated on 2024-02-14 03:10

Coverage Summary for Class: Info (org.springframework.boot.actuate.info)

Class	Method, %	Branch, %	Line, %
Info	28.6% (2/7)	0% (0/10)	25% (4/16)
Info$Builder	75% (3/4)	71.4% (5/7)	
Total	45.5% (5/11)	0% (0/10)	39.1% (9/23)

```java
/*
 * Copyright 2012-2022 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.boot.actuate.info;

import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Map;

import com.fasterxml.jackson.annotation.JsonAnyGetter;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonInclude.Include;

/**
 * Carries information of the application.
 * <p>
 * Each detail element can be singular or a hierarchical object such as a POJO or a nested
 * Map.
 *
 * @author Meang Akira Tanaka
 * @author Stephane Nicoll
 * @since 1.4.0
 */
@JsonInclude(Include.NON_EMPTY)
public final class Info {

	private final Map<String, Object> details;

	private Info(Builder builder) {
		Map<String, Object> content = new LinkedHashMap<>(builder.content);
		this.details = Collections.unmodifiableMap(content);
	}

	/**
	 * Return the content.
	 * @return the details of the info or an empty map.
	 */
	@JsonAnyGetter
	public Map<String, Object> getDetails() {
		return this.details;
	}

	public Object get(String id) {
		return this.details.get(id);
	}

	@SuppressWarnings("unchecked")
	public <T> T get(String id, Class<T> type) {
		Object value = get(id);
		if (value != null && type != null && !type.isInstance(value)) {
			throw new IllegalStateException("Info entry is not of required type [" + type.getName() + "]: " + value);
		}
		return (T) value;
	}

	@Override
	public boolean equals(Object obj) {
		if (obj == this) {
			return true;
		}
		if (obj instanceof Info other) {
			return this.details.equals(other.details);
		}
		return false;
	}

	@Override
	public int hashCode() {
		return this.details.hashCode();
	}

	@Override
	public String toString() {
		return getDetails().toString();
	}

	/**
	 * Builder for creating immutable {@link Info} instances.
	 */
	public static class Builder {

		private final Map<String, Object> content;

		public Builder() {
			this.content = new LinkedHashMap<>();
		}

		/**
		 * Record detail using given {@code key} and {@code value}.
		 * @param key the detail key
		 * @param value the detail value
		 * @return this {@link Builder} instance
		 */
		public Builder withDetail(String key, Object value) {
			this.content.put(key, value);
			return this;
		}

		/**
		 * Record several details.
		 * @param details the details
		 * @return this {@link Builder} instance
		 * @see #withDetail(String, Object)
		 */
		public Builder withDetails(Map<String, Object> details) {
			this.content.putAll(details);
			return this;
		}

		/**
		 * Create a new {@link Info} instance based on the state of this builder
```



```java
	 * Create a new {@link Info} instance based on the state of this builder.
	 * @return a new {@link Info} instance
	 */
	public Info build() {
		return new Info(this);
	}

	}
}
```

After incorporating this test case, the coverage metrics showed significant improvements in methods, lines, and branches within the tested class. However, to further enhance coverage, especially for untested branches and methods in the info class, additional tests could be considered. These might include:

- Testing the equal method with various expected outcomes.
- Calling the get method with both valid and invalid types.

Such tests would delve deeper into the framework's internal functionality rather than focusing solely on the actuator smoke test system.

Conclusion

This test case effectively enhances the test coverage of the Spring Boot Actuator's info endpoint, particularly focusing on the contribute method's functionality. For comprehensive coverage, further tests exploring additional branches and methods within the info class are recommended.