ＳＴＥＰ０９-２

Ｓｐｒｉｎｇフレームワーク入門

コードで理解するＳｐｒｉｎｇの世界

jUnit/Spring/Mockitoを使用したテスト

STEP09-2 Webアプリケーション全体をテストしてみる（結合テスト的）

目次

[1 テストカバレッジを可視化する 1](#_Toc467600335)

[1.1 EclEmma （Eclipseプラグイン） 導入 1](#_Toc467600337)

[1.2 EclEmmaでカバレッジレポートを出力する対象を設定する 1](#_Toc467600338)

[1.3 テストケースを実行してカバレッジレポートを出力する 1](#_Toc467600339)

[2 Webアプリケーション全体を一気にテストする 3](#_Toc467600340)

[2.1 メイン画面表示処理 （MainController） のテストケース（正常系）作成 3](#_Toc467600342)

[2.2 メイン画面表示処理 （MainController） のテストケース（正常系）実行 4](#_Toc467600343)

[2.3 プロジェクト全体のテストケースを一気に実行 4](#_Toc467600344)

[2.4 プロジェクト全体のテストカバレッジを確認する 4](#_Toc467600345)

[2.5 書籍一覧コントローラ （BookController） のテストケース（正常系）新規作成 5](#_Toc467600346)

[2.6 書籍一覧コントローラ （BookController） のテストケース実行 6](#_Toc467600347)

[2.7 プロジェクト全体のテストケースを一気に実行 6](#_Toc467600348)

[2.8 プロジェクト全体のテストカバレッジを確認する 6](#_Toc467600349)

[2.9 書籍登録コントローラ （BookController） のテストケース（正常系・異常系）作成 7](#_Toc467600350)

[2.10 書籍登録コントローラ （BookController） のテストケース実行 8](#_Toc467600351)

[2.11 プロジェクト全体のテストケースを一気に実行 8](#_Toc467600352)

[2.12 プロジェクト全体のテストカバレッジを確認する 8](#_Toc467600353)

[2.13 カバレッジ リポートを出力する 8](#_Toc467600354)

# テストカバレッジを可視化する

テストのカバレッジを可視化するツール「EclEmma」というプラグインをEclipseに導入する。



## EclEmma （Eclipseプラグイン） 導入

|  |
| --- |
| ① EclEmma のインストール  Help → Eclipse MarketPlace  検索画面で 「EclEmma」を検索して，以下のツールをインストールする．  -----------------  EclEmma Java Code Coverage 2.3.3  -----------------  プラグインは全て選択してインストール → 再起動 |

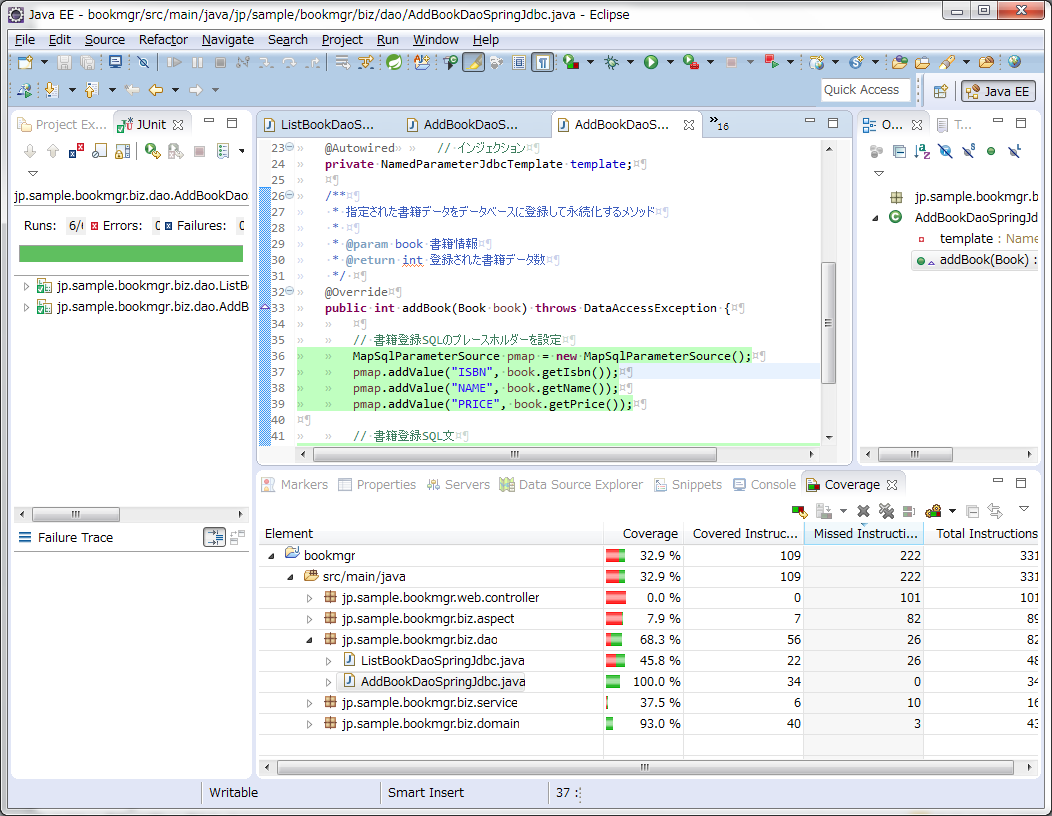
## テストケースを実行してカバレッジレポートを出力する

|  |
| --- |
| ① プロジェクト全体のテストケースを一気に実行して、その結果のカバレッジレポートを出力させる。  Package Explorerでプロジェクト「bookMgr」を選択して右クリック → Coverage As → JUnit Test  ② テストケース実行結果確認  EclipseのJUnitビューが表示され、テストを実行。  テスト実行が完了すると、その結果が表示される。  テストコードにエラーがある場合や、テスト結果にバグがある場合は、アイコンで識別できる。  また、エラーをダブルクリックすると、該当テストケースのコードが表示される。  ③ カバレッジレポートを確認する  EclipseのCoverageビューが表示されるので、カバレッジの内容を確認する。 |

## EclEmmaでカバレッジレポートを出力する対象を変更するには？

テストコードもカバレッジ対象となってしまう時があるので、そのような場合は、以下の操作でカバレーッジリポート対象範囲を変更することが可能。

|  |
| --- |
| ① カバレッジコンフィグレーション画面を出す。  Package Explorerでプロジェクト「bookMgr」を選択して右クリック → Coverage As → Coverage Configurations..  ② カバレッジレポート対象範囲を設定  Coverage タブを選択すると、候補となるスコープ（フォルダやライブラリ）一覧が表示される  今回の対象は、「bookMgr – src/main/java」 なので、そこだけにチェックを入れた状態で 「Apply」する |



# Webアプリケーション全体を一気にテストする

**STEP09-2では、擬似的なHTTPリクエストを発行して、作成したWebアプリケーション全体のテスト方法を学習します。**つまり Controller - Service - Dao を一気通貫に動作させてテストする方法を学習します。総合テストやシナリオテストに応用可能な手法です。



## メイン画面表示処理 （MainController） のテストケース（正常系）作成

１）メイン画面表示のテストケース（正常系の例）新規作成

|  |
| --- |
| /src/test/java/jp.sample.bookmgr.web.controller.MainControllerTest.java （新規作成） |
| package jp.sample.bookmgr.web.controller;  import static org.springframework.test.web.servlet.setup.MockMvcBuilders.\*;  import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  import org.junit.Before;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.test.context.ContextConfiguration;  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  import org.springframework.test.context.web.WebAppConfiguration;  import org.springframework.test.web.servlet.MockMvc;  import org.springframework.web.context.WebApplicationContext;  /\*\*  \* Mainコントローラクラスのテストケース  \*  \* @author 長住@NTT-AT  \* @version 1.0  \*/  // SpringによるJunitテストランナー  @RunWith(SpringJUnit4ClassRunner.class)  // WebApplicationContext をロードした環境下でテストコードを実行できるおまじない  @WebAppConfiguration  // Spring定義ファイルの取り込み  @ContextConfiguration({"classpath:test-context-biz.xml", "classpath:test-context-web.xml"})  public class MainControllerTest {  /\*\*  \* WebApplicationContext  \*/  @Autowired  private WebApplicationContext wac;  /\*\*  \* Spring MVCのテストをするためのMockMvcクラス  \*/  private MockMvc mockMvc;  /\*\*  \* 事前のおまじない  \* MockMvcクラスインスタンスを取得  \*/  @Before  public void setup() {  // MockMvcクラスインスタンスを取得  mockMvc = webAppContextSetup(wac).build();  }  /\*\*  \* メイン画面表示処理のテスト:正常系その１  \* GETリクエストを模倣し返って来たHTTPステータスコードとビューの名前をチェック  \*/  @Test  public void testMain\_normal\_1() throws Exception {  // メイン画面表示のGETリスクエスト発行  mockMvc.perform(get("/main"))  .andExpect(status().isOk())  .andExpect(view().name("main"));  }  } |

## メイン画面表示処理 （MainController） のテストケース（正常系）実行

テストケース（MainControllerTest.java）をjUnitで実行し、結果を確認する。

## プロジェクト全体のテストケースを一気に実行

プロジェクト全体のテストケースを一度に実行し、結果を確認する。

## プロジェクト全体のテストカバレッジを確認する

プロジェクト全体のテストケースを一度に実行し、その結果のカバレッジを確認する。

## 書籍一覧コントローラ （BookController） のテストケース（正常系）新規作成

１）書籍一覧のテストケース（正常系の例）新規作成

|  |
| --- |
| /src/test/java/jp.sample.bookmgr.web.controller.BookControllerTest.java （新規作成） |
| package jp.sample.bookmgr.web.controller;  import static org.junit.Assert.\*;  import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  import static org.springframework.test.web.servlet.setup.MockMvcBuilders.\*;  import java.util.List;  import org.hamcrest.CoreMatchers;  import org.junit.Before;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.test.context.ContextConfiguration;  import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  import org.springframework.test.context.web.WebAppConfiguration;  import org.springframework.test.web.servlet.MockMvc;  import org.springframework.test.web.servlet.MvcResult;  import org.springframework.test.web.servlet.ResultActions;  import org.springframework.ui.ModelMap;  import org.springframework.web.context.WebApplicationContext;  import jp.sample.bookmgr.biz.domain.Book;  /\*\*  \* Bookコントローラクラスのテストケース  \*  \* @author 長住@NTT-AT  \* @version 1.0  \*/  // SpringによるJunitテストランナー  @RunWith(SpringJUnit4ClassRunner.class)  // WebApplicationContext をロードした環境下でテストコードを実行できるおまじない  @WebAppConfiguration  // Spring定義ファイルの取り込み  @ContextConfiguration({"classpath:test-context-biz.xml", "classpath:test-context-web.xml"})  public class BookControllerTest {  /\*\*  \* WebApplicationContext  \*/  @Autowired  private WebApplicationContext wac;  /\*\*  \* Spring MVCのテストをするためのMockMvcクラス  \*/  private MockMvc mockMvc;    /\*\*  \* 事前のおまじない  \* MockMvcクラスインスタンスを取得  \*/  @Before  public void setup() {  // MockMvcクラスインスタンスを取得  mockMvc = webAppContextSetup(wac).build();  }    /\*\*  \* 書籍一覧画面表示処理のテスト:正常系その１  \* GETリクエストを模倣し返って来たHTTPステータスコードとビューの名前をチェック  \* DBに規定データが登録されていることを前提にDBから取得したデータが正しいかもチェック  \*/  @SuppressWarnings("unchecked")  @Test  public void testListBook\_normal\_1() throws Exception {  // 書籍一覧表示画面のGETリスクエスト発行  ResultActions retact = mockMvc.perform(get("/listbook"))  .andExpect(status().isOk()) // HTTPステータスコード  .andExpect(view().name("listbook")); // ビュー名  // DBから読み込んだ書籍情報リストを取り出す  MvcResult mvcResult = retact.andReturn();  ModelMap modelMap = mvcResult.getModelAndView().getModelMap();  List<Book> booksList = (List<Book>)modelMap.get("books");  assertThat(booksList, CoreMatchers.is(CoreMatchers.not(CoreMatchers.nullValue())));    // DBに ID=1、ISBN="123456789abcdefgh" NAME="JavaScript" PRICE=1200  // のデータが登録されていることが前提のテストケース  Book book1 = booksList.get(0);  assertThat(book1.getId(), CoreMatchers.is(1));  assertThat(book1.getIsbn(), CoreMatchers.is("123456789abcdefgh"));  assertThat(book1.getName(), CoreMatchers.is("JavaScript"));  assertThat(book1.getPrice(), CoreMatchers.is(1200));  }  } |

## 書籍一覧コントローラ （BookController） のテストケース実行

テストケース（BookControllerTest.java）をjUnitで実行し、結果を確認する。

## プロジェクト全体のテストケースを一気に実行

プロジェクト全体のテストケースを一度に実行し、結果を確認する。

## プロジェクト全体のテストカバレッジを確認する

プロジェクト全体のテストケースを一度に実行し、その結果のカバレッジを確認する。

## 書籍登録コントローラ （BookController） のテストケース（正常系・異常系）作成

１）書籍登録処理のテストケース（正常系の例）追加

|  |
| --- |
| /src/test/java/jp.sample.bookmgr.web.controller.BookControllerTest.java |
| ● 93行目付近 |
| 93  /\*\*  \* 書籍登録処理のテスト:正常系その１  \* 書籍フォームからのPOSTリクエストを模倣し実際にデータベースに登録させる。  \* また、返って来たHTTPステータスコードとビューの名前をチェックする。  \* @throws Exception  \*/  @Test  public void testAddBook\_normal\_1() throws Exception {    // テストで登録する書籍情報を設定  Book book = new Book();  book.setIsbn("TEST-1234567890");  book.setName("testAddBookで自動登録しました");  book.setPrice(8888);  // 書籍登録POSTリクエスト発行  mockMvc.perform(post("/addbook")  .param("isbn", book.getIsbn())  .param("name", book.getName())  .param("price", new Integer(book.getPrice()).toString())  )  .andExpect(status().isFound()) // HTTPステータスコード  .andExpect(view().name("redirect:result")) // ビュー名  .andExpect(model().hasNoErrors()); // エラーがないこと  } |

２）書籍登録処理のテストケース（異常系）追加

|  |
| --- |
| /src/test/java/jp.sample.bookmgr.web.controller.BookControllerTest.java |
| ● 119行目付近 |
| 119  /\*\*  \* 書籍登録処理のテスト:異常系その１  \* 書籍フォームからのPOSTリクエストを模倣するがバリデーションエラーとなるケース。  \* 返って来たHTTPステータスコードとビューの名前をチェックする。  \* @throws Exception  \*/  @Test  public void testAddBook\_abnormal\_1() throws Exception {    // テストで登録する書籍情報を設定  Book book = new Book();  book.setIsbn("1234"); // @Size 違反  book.setName("testAddBookで自動登録しました");  book.setPrice(8888);  // 書籍登録POSTリクエスト発行  mockMvc.perform(post("/addbook")  .param("isbn", book.getIsbn())  .param("name", book.getName())  .param("price", new Integer(book.getPrice()).toString())  )  .andExpect(status().isOk()) // HTTPステータスコード  .andExpect(view().name("addbookform")) // ビュー名  .andExpect(model().hasErrors()) // エラーが発生していること  .andExpect(model().errorCount(1)) // エラーの数  .andExpect(model().attributeExists("book")) // modelAtribute名  .andExpect(model().attributeHasFieldErrors("book", "isbn")) // エラーフィールド名  ;  } |

## 書籍登録コントローラ （BookController） のテストケース実行

テストケース（BookControllerTest.java）をjUnitで実行し、結果を確認する。

## プロジェクト全体のテストケースを一気に実行

プロジェクト全体のテストケースを一度に実行し、結果を確認する。

## プロジェクト全体のテストカバレッジを確認する

プロジェクト全体のテストケースを一度に実行し、その結果のカバレッジを確認する。

## カバレッジ リポートを出力する

カバレッジビューで、右クリック　→ Export Session → Java → Coverage Report

レポートを出力するディレクトリを指定して出力させる。

HTML形式で詳細なカバレーッジレポートが出力されるのでブラウザで確認してみる。