

# Spring Bootアプリケーションのテスト

Ver3.2.1-2

# 本研修について

# アジェンダ

- ・想定している主な受講者
- ・本研修の目標
- ・前提知識
- ・研修のアウトライン
- ・本研修で登場するプログラムの役割
- ・本研修で使用する主な製品のバージョン

# 想定している主な受講者

- Spring Bootを使用する開発プロジェクトにこれから携わる、もしくは既に携わっている方
  - テストコードをほとんど書いたことがないので知識を得たい
  - テストコードの書き方を体系的に知りたい

# 本研修の目標

- ・自動テストのテストレベルと、テスト対象について説明できる
- ・JUnitを使って、Spring Bootアプリケーションを体系的にテストできる
- ・MockMvcを活用してテストできる
- ・Mockitoを使用してモックを活用できる

# 前提知識

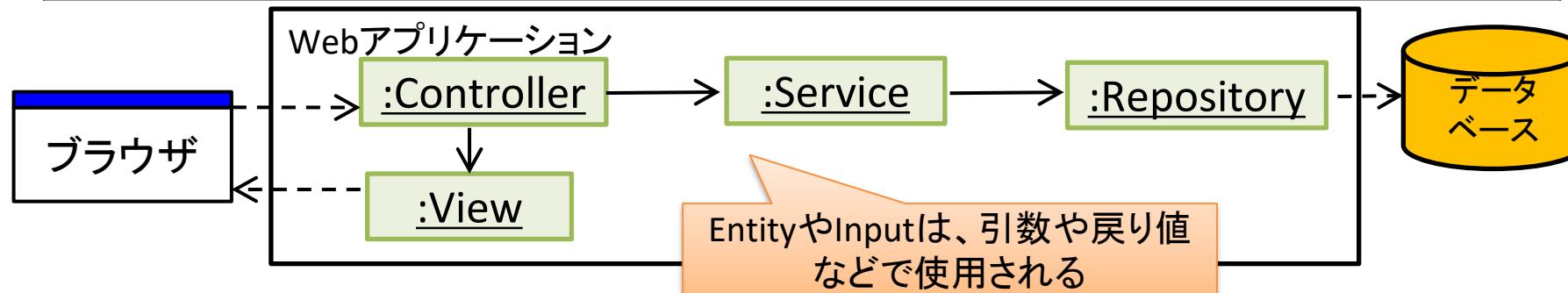
- Spring Bootを使用したアプリケーションの開発に従事したことがある、もしくは、研修等で学習したことがある

# 研修のアウトライン

- 自動テストの種類
- Springのテストサポート
- Repositoryのユニットテスト
- ServiceのユニットテストとMockitoの使い方
- Service・Repositoryのインテグレーションテスト
- ControllerのユニットテストとMockMvcの使い方
- Controller・Service・Repositoryのインテグレーションテスト
- RESTful Webサービスのテスト
- Spring Securityのテストサポート
- Selenideを用いたE2Eテスト
- フラッシュスコープの扱い
- セッションスコープのBeanの扱い

# 本研修で登場するプログラムの役割

役割名	説明
Controller	リクエストを受け取って業務ロジックを呼び出し、適切なViewを呼び出すまでの全体の処理の流れを制御する
View	動的なデータを埋め込みながらHTMLを作成する。動的なデータには、例えばデータベースから検索したデータがある
Service	業務ロジックを行う。業務ロジックは、例えば、在庫数をチェックして、在庫が足りていれば注文データを作成し、足りなければ例外をスローするというような処理
Repository	データベースにアクセスしてデータを取得したり保存したりする
Entity	業務データ(商品データなど)を保持する。データベースのテーブルが既に存在する場合は、テーブルと対にして作成することが多い。例えば、商品テーブルに対して、ProductクラスをEntityクラスとして作成するイメージとなる
Input	ユーザが入力した情報(注文時の名前・住所など)を保持する



# 本研修で使用する主な製品のバージョン

製品	バージョン
Java	17
Spring Framework	6.1.2
Spring Boot	3.2.1
Spring Security	6.2.1
Thymeleaf	3.1.2
JUnit	5.10.1

# 自動テストの種類

# この章の目標

- ・自動テストの意味とメリットを理解する
- ・Webアプリケーションの自動テストの種類を把握する

# 手動テストと自動テスト

- 手動テスト
  - 人間の手でプログラムを動かして、結果を目視で確認
    - 通常は画面を操作しながらテストするため、打鍵テストとも呼ばれる
- 自動テスト
  - テストコードがプログラムを動かして結果を確認
    - テストコードがプロダクションコード（本番で動くコード）のメソッドを呼び出し、戻り値の値が期待通りか？データベースの値が期待通りに変更されてるか？といった確認をプログラムで行う
    - 自動テストの種類によっては、テストコードがブラウザの操作を自動的に行って、画面の表示内容を確認
    - Javaアプリケーションの場合、一般的にJUnitを使用する

補足あり

# 自動テストのメリット

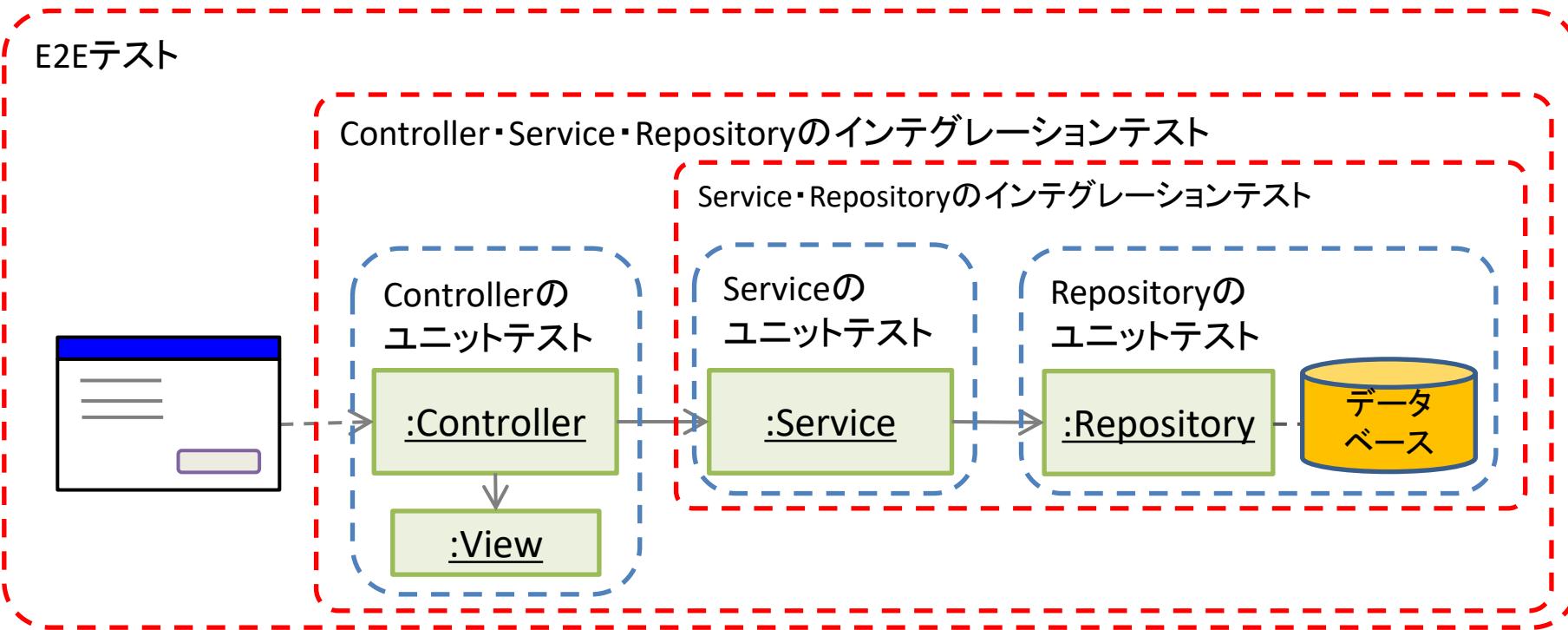
- 回帰テストの実施が容易になる
  - 回帰テストとは?
    - これまで動いていた機能が、プログラムに変更を加えた後も、変わらず動くことを確認するテスト
    - 手動テストで回帰テストする場合は、テストの実施に工数がかかりてしまうため、回帰テストが疎かになりがち
    - 自動テストであれば、テストコードさえ用意できれば、回帰テストを容易に実施できる
  - 作成したプログラムを部分的にすぐに動かすことができる
    - 例えば、データベースアクセスする部分を作成したら、そこだけ動かすことができる

# 自動テストの代表的なテストレベル

- ユニットテスト(単体テスト)
  - 1つの処理を単独でテストする
- インテグレーションテスト(結合テスト)
  - 複数の処理を繋げてテストする
- E2Eテスト(エンド・ツー・エンドテスト)
  - アプリケーション全体(端から端まで)を繋げてテストする

開発プロジェクトによって呼び方や意味が変わることがある

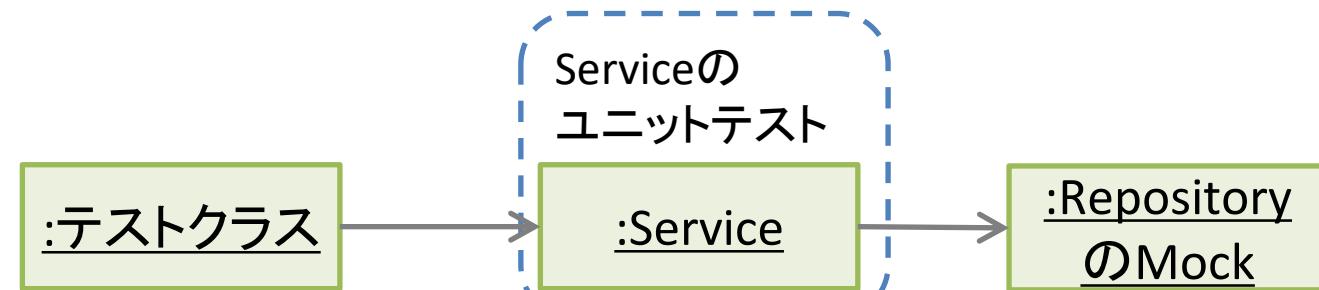
# Webアプリケーションの自動テストの主な種類



効率や工数などを加味しながら、どの種類をどれくらい網羅的にやるのかを、各開発プロジェクトで検討する

# ユニットテストとMock

- Mockとは
  - テスト用の偽物のクラスやそのオブジェクトのこと
  - テストするのに都合の良い値を返したり、呼ばれたメソッドを記録して呼ばれたことを検証可能にしたりする



- Repositoryが完成してなくてもServiceのテストができるし、Mockで任意の戻り値を返せるのでテストパターンの網羅もしやすい

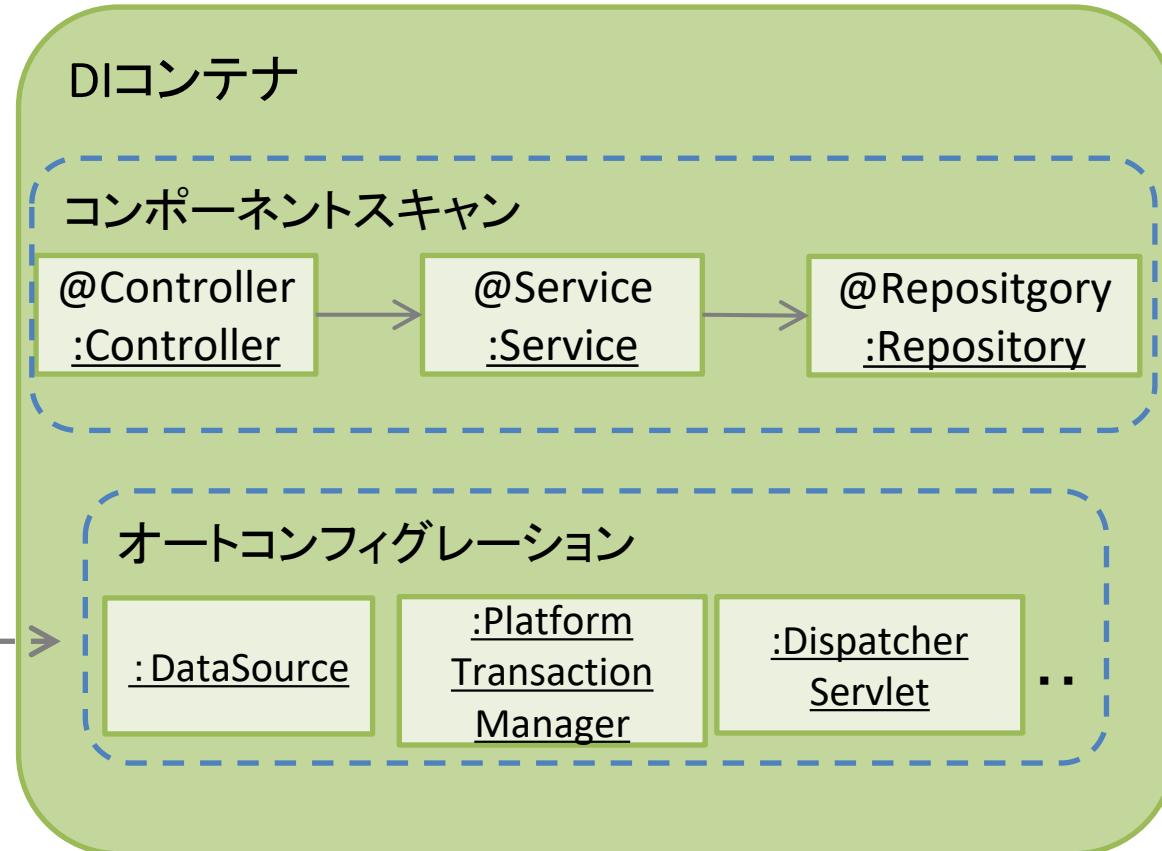
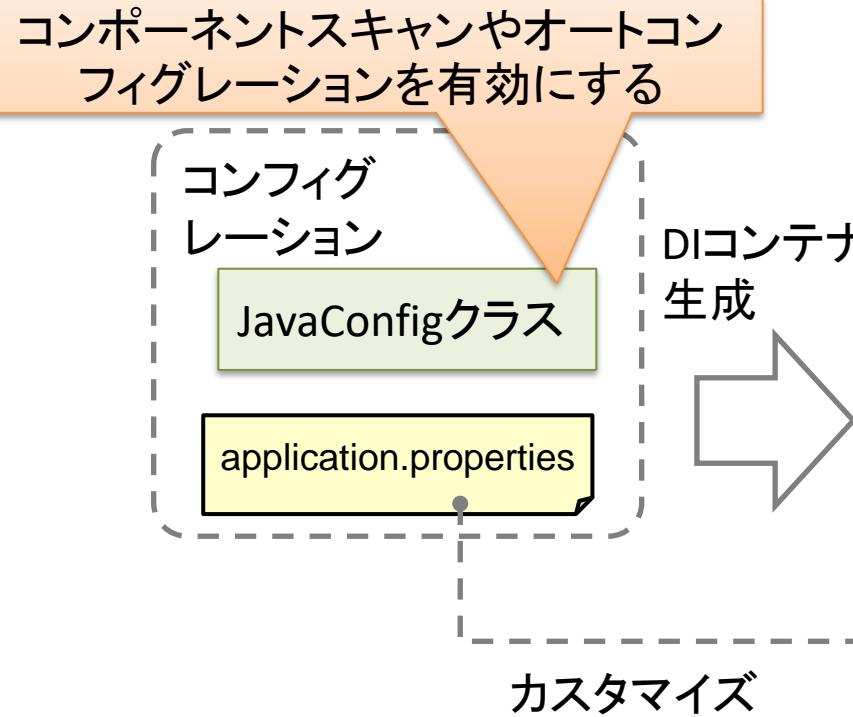
後述するMock用のライブラリを使用すれば、簡単にMockのオブジェクトを作成できる

# Springのテストサポート

# この章の目標

- Spring Bootアプリケーションのテストコードのイメージを掴む
- DIコンテナを生成するためのアノテーションの種類を知る
- JavaConfigクラスがどのように探されるかを理解する

# Spring Bootアプリケーションの作り



- ・自動テストの際も、基本的にはDIコンテナを起動する
- ・テストしたいBeanを取得して、メソッドを呼び出して挙動をテストする

# Springのテストサポート機能

## 【主な機能】

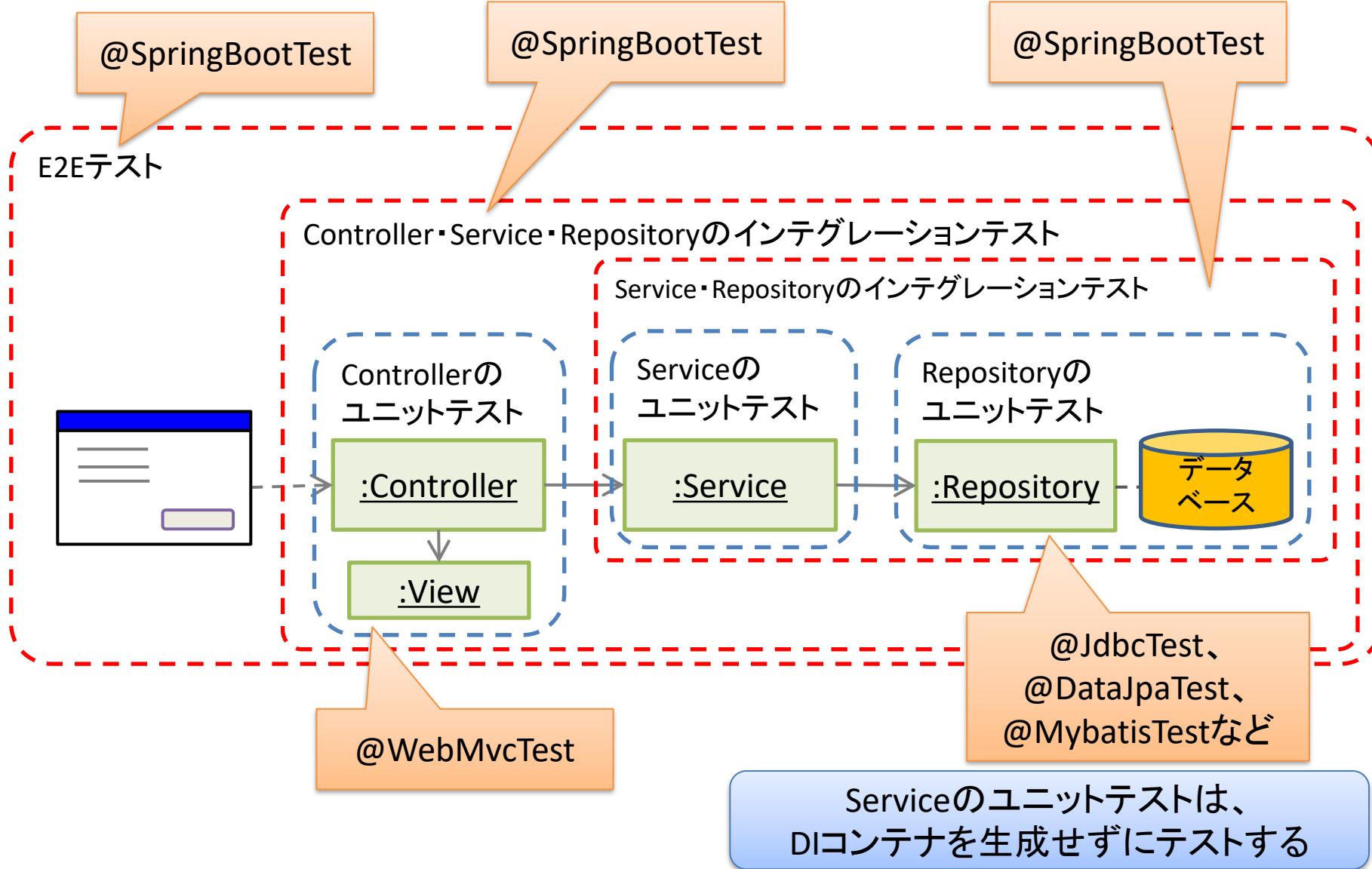
- 自動的にDIコンテナを生成
  - JUnitのテストを実行すると、自動的にDIコンテナが生成される
  - オートコンフィグレーションやコンポーネントスキャンを制限し、テストに不要なBeanの生成を防ぐ
- テストクラスに任意のBeanをインジェクション
  - テスト対象のメソッドを持ったBeanをテストクラスにインジェクションし、テストメソッドの中で呼び出せる
- 指定したSQLファイルをテスト時に読み込んで実行
  - 任意のSQLファイルにSQLを記述し、テスト時に自動的に読み込んで実行してもらう。テストデータを用意する目的で使用
- データベースを自動的にロールバック
  - テストメソッドが終了するタイミングで、データベースのトランザクションのロールバックを自動的に行う。テストによって更新されたデータをテスト前の状態に戻すことで、次のテストに影響がないようにする

# DIコンテナを生成するためのアノテーション

- テストの種類によって使い分ける
  - @SpringBootTest
    - 通常のDIコンテナと基本的に同じ挙動のDIコンテナを生成。E2Eテストやインテグレーションテストで使用
  - @WebMvcTest
    - Web周りのコンフィグレーションだけ行うように、オートコンフィグレーションやコンポーネントスキャンを制限してDIコンテナを生成。Controllerのユニットテストを行うときに使用。
  - @JdbcTest、@DataJpaTest、@MybatisTestなど
    - データベースアクセスの仕組み(JDBC、JPA、MyBatisなど)毎にアノテーションが用意されている。データベースアクセス周りのコンフィグレーションだけ行うように、オートコンフィグレーションやコンポーネントスキャンを制限してDIコンテナを生成。Repositoryのユニットテストを行うときに使用

いすれのアノテーションも、JUnitの@ExtendWithを含んでおり、  
テスト実行時にDIコンテナ生成の処理を挟み込んでいる

# DIコンテナを生成するためのアノテーション



# テストクラスの記述イメージ

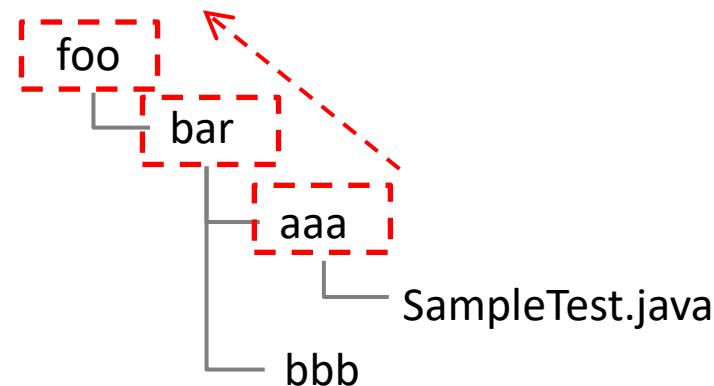
```
@SpringBootTest ...①
class SampleTest {
    @Autowired
    SomeBean someBean; ...②

    @Test
    void test() {
        someBean.foo(); ...③
        ...
    }
}
```

- |   |  |
|---|--|
| ① | DIコンテナを生成するためのアノテーション。読み込むJavaConfigクラスを明示的に指定することもできるが、ここでは省略している。省略した場合は、Springが自動的に探してくれる(後述) |
| ② | テスト対象のSomeBeanオブジェクトをインジェクション  |
| ③ | インジェクションしたオブジェクトのメソッドを呼び出してテストを実施  |

# JavaConfigクラスを自動的に探してくれる

- @Configurationではなく、@SpringBootConfiguration(@Configurationを含んでいる)が付いたJavaConfigクラスを探す
- テストクラスと同じパッケージの中を探し、見つからなければ、上位のパッケージの中を探し、そこでも見つからなければ、さらに上位のパッケージを探す



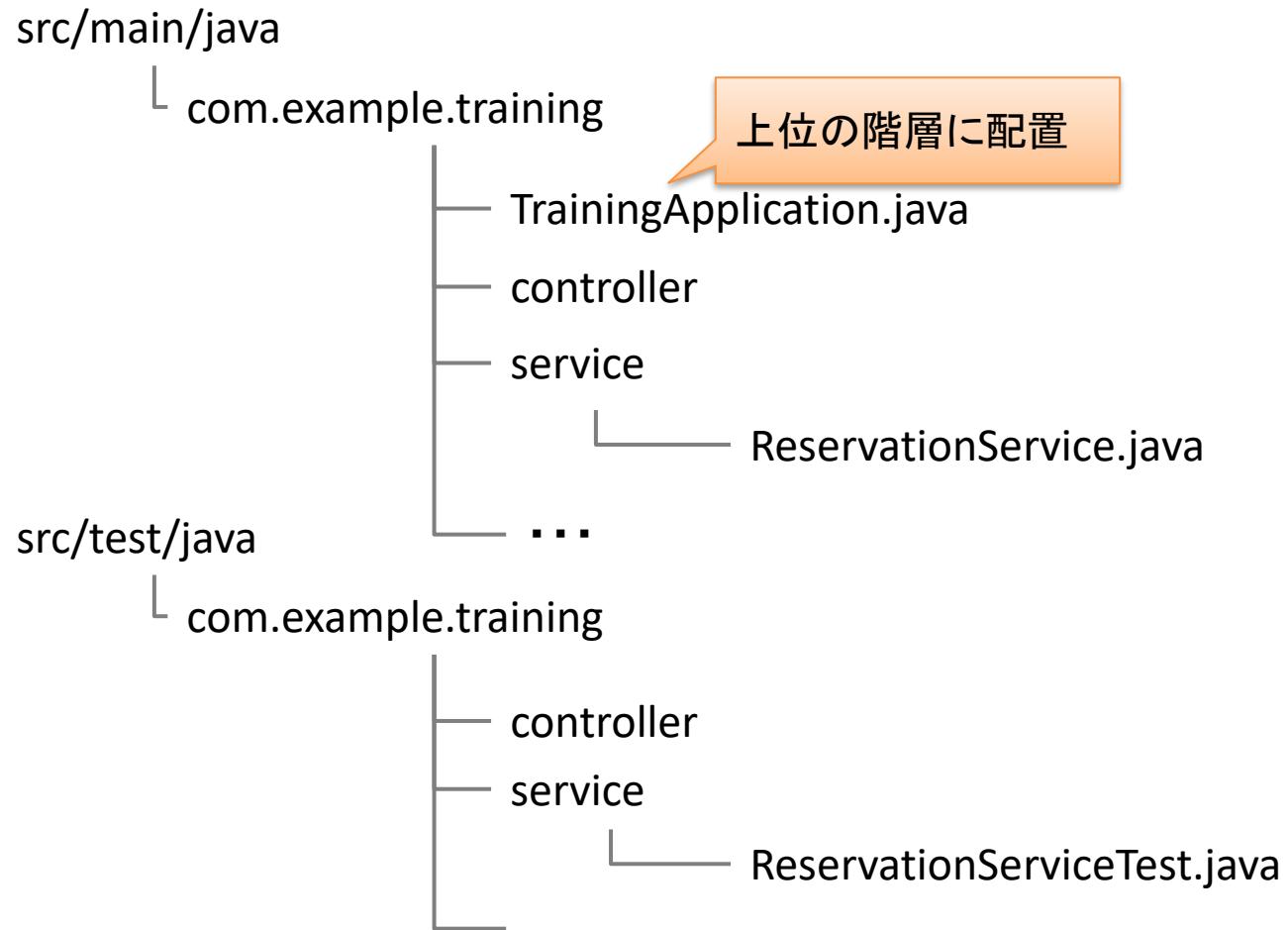
# @SpringBootConfigurationが付いたJavaConfigクラス

- Spring Bootのアプリケーションでは、通常、mainメソッドを持ったクラスに@SpringBootConfigurationが付いている

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

@SpringBootApplicationの中には@SpringBootConfigurationが含まれている。  
※オートコンフィグレーションを有効にする@EnableAutoConfiguration、コンポーネントスキャンを指示する@ComponentScanも含まれる

# @SpringBootConfigurationが付いたJavaConfigクラス



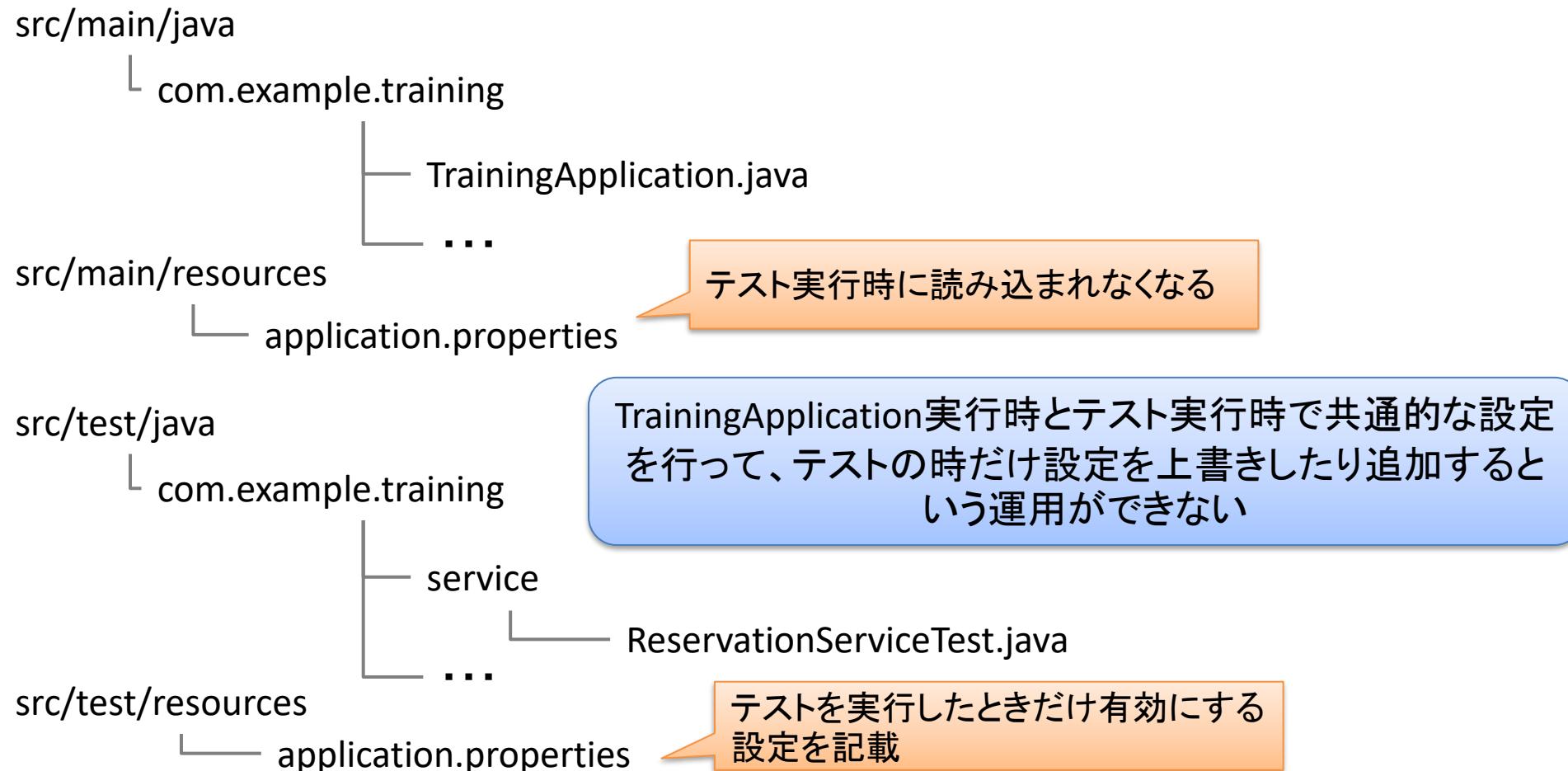
本番で動かすときと同じJavaConfigクラスを読み込むことで、テスト  
の時は動くのに本番だと動かない、といった現象を軽減

## テストを実行したときだけ読み込まれるapplication.properties

- テストのときだけ有効にしたい設定を行えると便利
  - テスト用のデータベースをコンフィグレーションする
  - ログレベルの設定を変える
  - など

# テストを実行したときだけ読み込まれるapplication.properties

- 案①: src/test/resourcesの直下にapplication.propertiesを作成



# テストを実行したときだけ読み込まれるapplication.properties

- 案②: 「default」プロファイルを利用して、テスト用の application-default.properties を配置  
補足あり

src/main/java

└ com.example.training

  └ TrainingApplication.java  
  ...  
  ...

src/main/resources

└ application.properties

TrainingApplication 実行時、テスト  
実行時の共通の設定を記載

src/test/java

└ com.example.training

  └ service  
    └ ReservationServiceTest.java  
  ...  
  ...

src/test/resources

└ application-default.properties

(プロファイルを指定せずに) テストを実行  
したときだけ有効にする設定を記載

# ハンズオン

---

- ・「1901-training-test-overview」

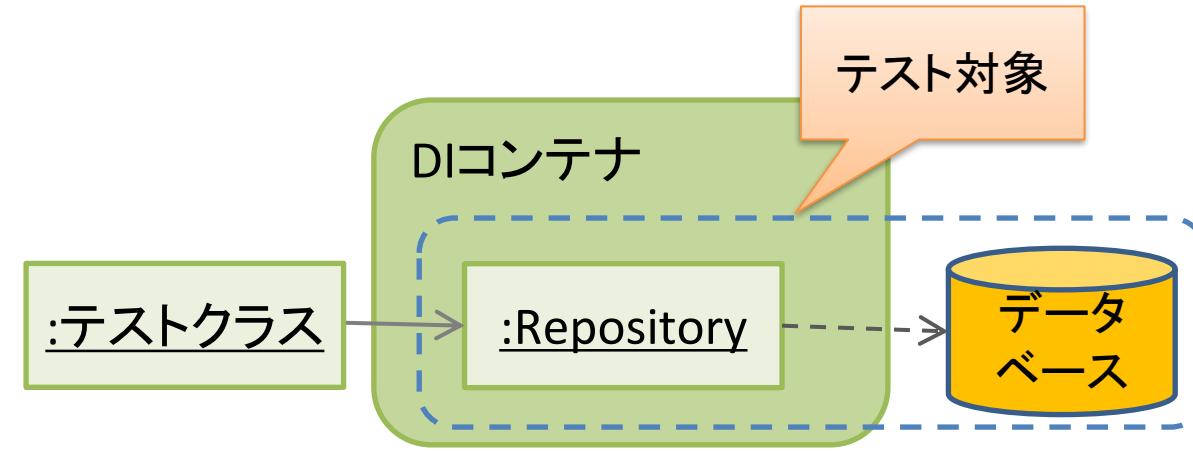
# **Repositoryのユニットテスト**

# この章の目標

---

- Repositoryのユニットテストを作成できる
- テストデータの登録とクリーンができる
- 更新系の処理のテストができる

# Repositoryのユニットテストのイメージ



# DIコンテナを生成するためのアノテーション

- @JdbcTest、@DataJpaTest、@MybatisTestなどデータアクセスの仕組み毎に用意されている
  - 本研修では、RepositoryがSpring JDBC(JdbcTemplateクラス)を使用している想定なので、@JdbcTestを使用
- オートコンフィグレーションとコンポーネントスキャンに制限をかけて、データベースアクセスに関するコンフィグレーションだけ有効にしてDIコンテナを生成
  - 無駄なコンフィグレーションが行われずテストの実行が速くなる

# テスト対象のクラス(プロダクションコード)

```
@Repository  
public class JdbcTrainingRepository implements TrainingRepository {  
  
    private final JdbcTemplate jdbcTemplate;  
  
    public JdbcTrainingRepository(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
  
    @Override  
    public Training selectById(String id) {  
        return jdbcTemplate.queryForObject("SELECT * FROM training WHERE id=?",  
            new DataClassRowMapper<>(Training.class),  
            id);  
    }  
    ...
```

JdbcTemplateオブジェクトをインジェクション

# テストクラスのサンプル

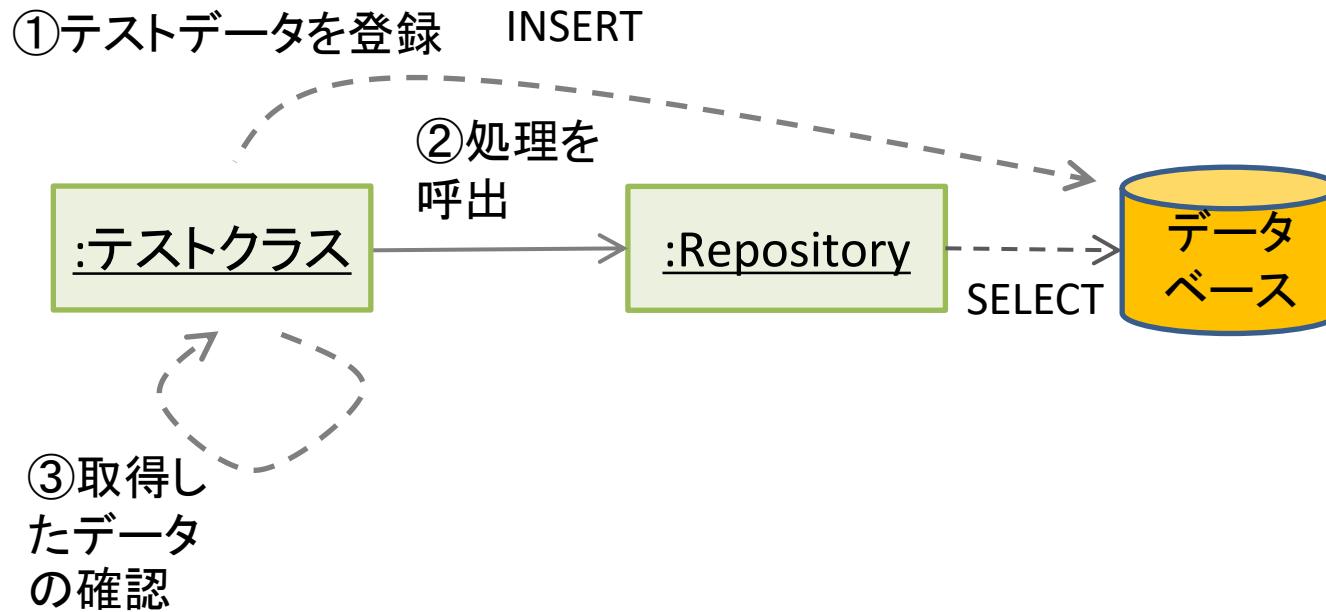
```
@JdbcTest ...①
@Import(JdbcTrainingRepository.class) ...②
class JdbcTrainingRepositoryTest {
    @Autowired
    TrainingRepository trainingRepository; ...③

    @Test
    void test_selectById() {
        Training training = trainingRepository.selectById("t01"); ...④
        assertThat(training.getTitle()).isEqualTo("ビジネスマナー研修"); ...⑤
    }
    ...
}
```

①	テストを実行すると自動的にDIコンテナが生成される。読込むJavaConfigクラスは自動的に探してくれる。コンポーネントスキャンを制限するため、@Repositoryが付いたRepositoryクラスのBeanは自動的に登録はされない
②	@Importは、ステレオタイプアノテーションが付いたクラスを個別に読み込むためのアノテーション。JdbcTrainingRepositoryクラス(@Repositoryが付いている)のオブジェクトがBeanとして登録される
③	@Autowiredでインジェクション
④	テスト対象であるRepositoryオブジェクトのメソッドを呼び出す
⑤	呼び出したメソッドの戻り値が期待通りになっているかをアサーション

# データベースのデータの用意

- テスト用のデータの用意もテストコードで行う



補足あり

データベースは、デフォルトだと組み込みデータベースが使われる

# @Sqlアノテーション

- @Sqlをテストメソッドに付けて、()括弧の中にSQLが記述されたファイルのパスを指定すると、テストメソッドが実行される直前に、記述されているSQLが実行される

```
@Test  
@Sql("JdbcTrainingRepositoryTest.sql")  
void test_selectById() {  
    Training training = trainingRepository.selectById("t01");  
    assertThat(training.getTitle()).isEqualTo("ビジネスマナー研修");  
}
```

JdbcTrainingRepositoryTest.sql

```
INSERT INTO training  
(id, title, start_date_time, end_date_time, reserved, capacity) VALUES  
('t01', 'ビジネスマナー研修', '2023-08-01 09:30', '2023-08-03 17:00', 1, 10)  
,'t02', 'Java実践', '2023-09-01 09:30', '2023-09-03 17:00', 1, 5)  
,'t03', 'マーケティング研修', '2023-10-01 09:30', '2023-10-03 17:00', 5, 5)  
;
```

# @Sqlアノテーション

- ファイルのパスは相対パスで指定できる

src/test/java

└ com.example.training.repository

└ JdbcTrainingRepositoryTest.java

src/test/resources

└ com.example.training.repository

└ JdbcTrainingRepositoryTest.sql

```
@Test  
@Sql("JdbcTrainingRepositoryTest.sql")  
void test_selectById() {  
    ...  
}
```

# @Sqlアノテーション

- SQLファイルを複数指定することも可能
  - 指定した順番に実行される

```
@Test  
@Sql({"JdbcTrainingRepositoryTest_1.sql", "JdbcTrainingRepositoryTest_2.sql"})  
void test_selectById() {  
    ...  
}
```

# @Sqlアノテーション

```
@JdbcTest  
@Import(JdbcTrainingRepository.class)  
@Sql("JdbcTrainingRepositoryTest.sql") ・・・①  
class JdbcTrainingRepositoryTest {  
    @Autowired  
    TrainingRepository trainingRepository;  
  
    @Test  
    void test_selectById() { ・・・②  
        Training training = trainingRepository.selectById("t01");  
        assertThat(training.getTitle()).isEqualTo("ビジネスマナー研修");  
    }  
  
    @Test  
    @Sql("JdbcTrainingRepositoryTest_2.sql")  
    void test_selectAll() { ・・・③  
        List<Training> trainings = trainingRepository.selectAll();  
        assertThat(trainings.size()).isEqualTo(3);  
    }  
}
```

クラスに付けることも可能

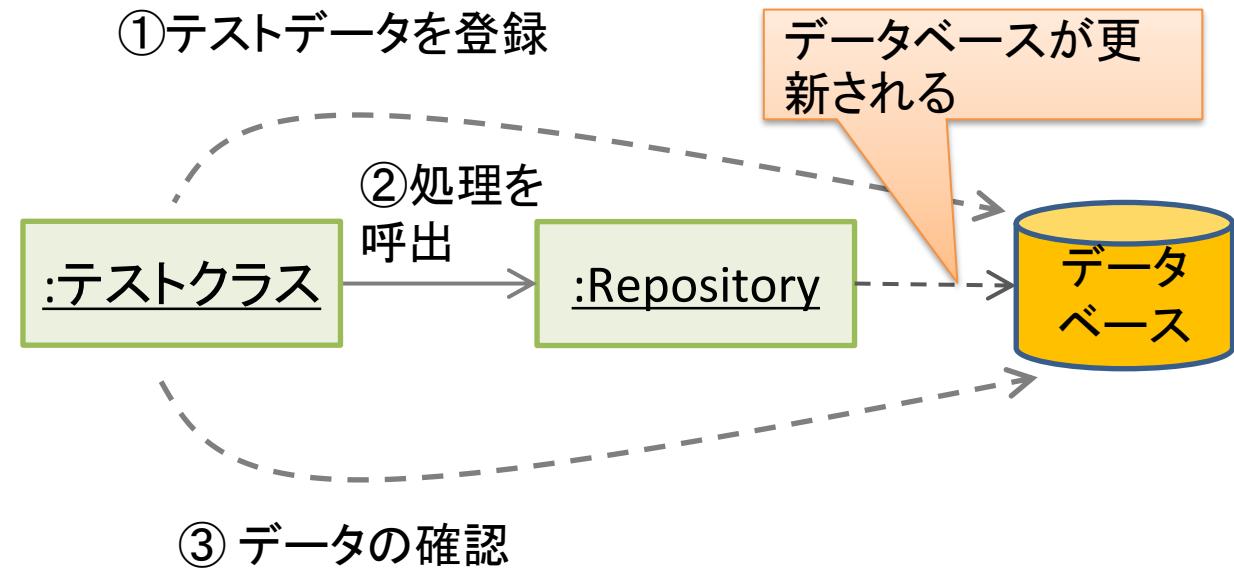
- |   |  |
|---|--|
| ① | @Sqlをクラスに付けると、指定したSQLファイルがすべてのテストメソッドで実行される  |
| ② | @Sqlが付いていないテストメソッドが呼びだされた際は、①で指定した JdbcTrainingRepositoryTest.sql のSQLが実行される           |
| ③ | @Sqlが付いているテストメソッドは、テストメソッドに付いてる@Sqlが指定している JdbcTrainingRepositoryTest_2.sql のSQLが実行される |

# データのクリーン

- テストメソッドが実行されると、そのテストメソッドで必要なデータが登録されるが、次のテストメソッドが呼ばれるときには、次のテストメソッドで必要なデータを登録し直すため、一旦データをクリーン(消す)する必要がある
- @JdbcTestを使った場合は、テストメソッドを実行するときに自動的にトランザクションが開始しされ、テストメソッドが終了すると自動的にロールバックされる
  - @Sqlで登録したデータもロールバックされてクリーンされる
  - これにより、次のテストに影響がない
- ロールバックの機能は、@JdbcTestに含まれている@Transactionalによって有効になっている

```
...
@Transactional
...
public @interface JdbcTest {
    ...
}
```

# 更新系の処理のテスト



# テスト対象のクラス(プロダクションコード)

```
@Repository
public class JdbcTrainingRepository implements TrainingRepository {

    private final JdbcTemplate jdbcTemplate;

    public JdbcTrainingRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    ...
    @Override
    public boolean update(Training training) {
        int count = jdbcTemplate.update("UPDATE training SET title=?,
                                         start_date_time=?,
                                         end_date_time=?,
                                         reserved=?,
                                         capacity=? WHERE id=?",
                                         training.getTitle(),
                                         training.getStartTime(),
                                         training.getEndTime(),
                                         training.getReserved(),
                                         training.getCapacity(),
                                         training.getId());
        return count > 0;
    }

    ...
}
```

# JdbcTemplateを使ったデータの確認

```
...
@Autowired
JdbcTemplate jdbcTemplate; ...①
...
@Test
void test_update() {
    Training training = new Training();
    training.setId("t01");
    training.setTitle("SQL入門");
    ...
    boolean result = trainingRepository.update(training); ...②
    assertThat(result).isEqualTo(true);
    Map<String, Object> trainingMap = jdbcTemplate.queryForMap( ...③
        "SELECT * FROM training WHERE id=?", "t01");
    assertThat(trainingMap.get("title")).isEqualTo("SQL入門"); ...④
    ...
}
```

データの確認方法は、今回のように、使用しているデータアクセスの仕組みが提供するクラスを使ったり(Spring JDBC の場合はJdbcTemplate)、DbUnitのようなライブラリを使うことも可能

①	JdbcTemplateオブジェクトをインジェクション
②	研修データを更新するメソッドを呼び出し、trainingテーブルのデータが更新される
③	JdbcTemplateオブジェクトを使って、更新されたはずのレコードをSELECT文で取得
④	取得したレコードのカラムの値をアサーション

# ハンズオン

---

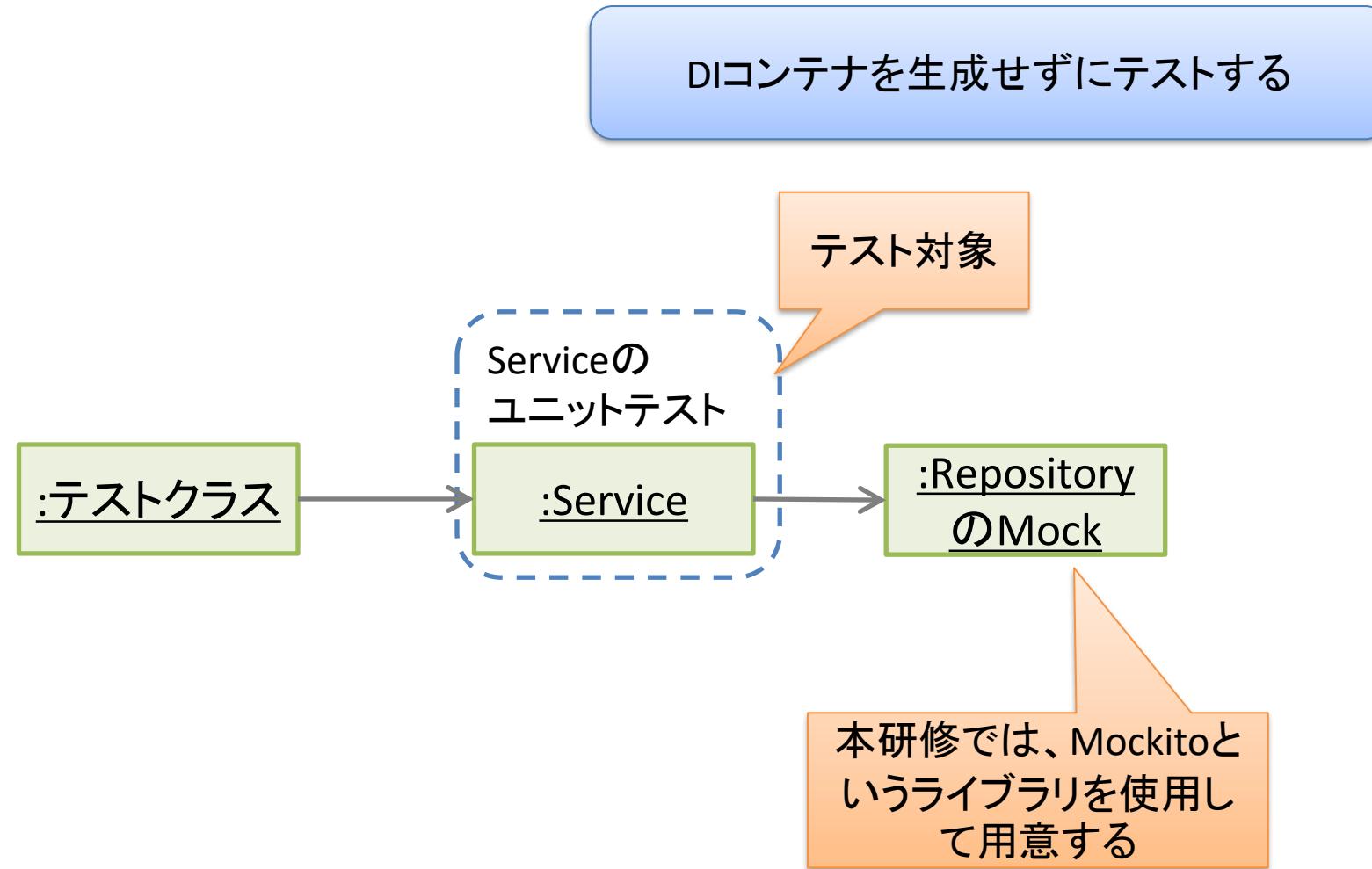
- ・「1952-shopping-test-repository」

# Serviceのユニットテスト

# この章の目標

- Serviceのユニットテストを作成できる
- Mockitoを利用してMockの挙動を指定できる
- Mockitoを利用して、Mockのメソッドに渡された引数の確認ができる

# Serviceのユニットテストのイメージ



# Mockitoの紹介

- Mockのオブジェクトを簡単に作成することができるオープンソースのライブラリ
- Mock用のライブラリとしてMockito以外のライブラリもあるが、Spring Bootを使用したアプリケーション開発では、特段の理由が無ければMockitoを使うのが一般的
  - Spring Bootのテスト用のStarters(spring-boot-starter-test)に含まれている
  - Mockitoと連動する機能をSpring Bootが提供している

# Mockitoを使用したServiceのユニットテストのサンプル

- テスト対象のServiceのクラス

```
@Service  
@Transactional  
public class TrainingAdminServiceImpl implements TrainingAdminService {  
  
    private final TrainingRepository trainingRepository;  
  
    public TrainingAdminServiceImpl(TrainingRepository trainingRepository) {  
        this.trainingRepository = trainingRepository;  
    }  
  
    @Override  
    public Training findById(String trainingId) {  
        return trainingRepository.selectById(trainingId);  
    }  
    ...  
}
```

RepositoryのselectByIdメソッドの戻り  
値が返されることをテストしたい

# Mockitoを使用したServiceのユニットテストのサンプル

```
@ExtendWith(MockitoExtension.class) ...①
class TrainingAdminServiceImplTest {
    @InjectMocks ...②
    TrainingAdminServiceImpl trainingAdminService;

    @Mock ...③
    TrainingRepository trainingRepository;

    @Test
    void test_findById() {
        Training training = new Training();
        training.setTitle("ビジネスマナー研修");
        Mockito.doReturn(training).when(trainingRepository).selectById("t01"); ...④

        Training actual = trainingAdminService.findById("t01"); ...⑤
        assertThat(actual.getTitle()).isEqualTo("ビジネスマナー研修"); ...⑥
    }
}
```

Springのアノテーションは  
使っていない

:TrainingAdmin  
ServiceImpl

:Mockの  
TrainingRepository

# Mockitoを使用したServiceのユニットテストのサンプル



- |   |  |
|---|--|
| ① | Mockitoが提供するMockitoExtensionクラスを指定し、テスト実行時に MockitoExtensionクラスの処理が呼び出され、②や③の行のようなMockitoが提供するアノテーションを検知し、アノテーションに従った処理を行ってくれる |
| ② | TrainingAdminServiceImplクラスのコンストラクタをMockitoが呼び出して、オブジェクトを自動的に生成しフィールドに代入される。その際、Mockオブジェクトとして用意された依存オブジェクトをインジェクションしてくれる       |
| ③ | TrainingRepository型のMockのオブジェクトが生成されフィールドに代入される  |
| ④ | Mockオブジェクトに挙動を指定。selectByIdメソッドが呼ばれたら、「ビジネスマナー研修」のTrainingオブジェクトを返す挙動を指定している   |
| ⑤ | テスト対象のServiceオブジェクトのメソッドを呼び出す。findByIdメソッドの中でMockオブジェクトのselectByIdメソッドが呼び出される想定  |
| ⑥ | 返されたTrainingオブジェクトのタイトルをアサーション   |

# Mockitoとstaticインポート

staticインポートしない場合

```
Mockito.doReturn(training).when(trainingRepository).selectById(Mockito.anyString());
```

staticインポートの宣言

```
import static org.mockito.Mockito.*;
```

staticインポートした場合

```
doReturn(training).when(trainingRepository).selectById(anyString());
```

# «参考»BDDスタイルの書き方

- BDD(Behavior Driven Development)のスタイルで書くことも可能
  - BDD:プログラムの振る舞いと結果を分かりやすく記述するための考え方

```
import static org.mockito.BDDMockito.*;
```

```
doReturn(training).when(trainingRepository).selectById(anyString());
```



```
when(trainingRepository.selectById(anyString())).willReturn(training);
```

本研修ではBDDスタイルは  
使用しない

# Mockの挙動を指定

- 戻り値を返す挙動

```
doReturn(training).when(trainingRepository).selectById("t01");
```

- 例外をスローする挙動

```
doThrow(DuplicateKeyException.class).when(trainingRepository).insert(training);
```

- «参考» 戻り値がvoidの挙動(基本、使うことはない)

```
doNothing().when(trainingRepository).insert(training);
```

# Mockのメソッドの引数を指定

```
doReturn(training).when(trainingRepository).selectById("t01");
```

引数で"t01"が指定された時だけ、trainingのオブジェクトを返す。"t01"以外の引数が指定された場合は、nullが返される

```
doReturn(training).when(trainingRepository).selectById(any());
```

どのような型・値(nullも含む)の引数が渡されても  
trainingオブジェクトを返す

```
doReturn(training).when(trainingRepository).selectById(anyString());
```

String型のどのような値(nullは含まない)の引数が渡されてもtrainingオブジェクトを返す。同等のメソッドで、anyIntやanyDoubleといったメソッドもある

# 複数回呼び出されるメソッドを指定

```
doReturn(training1).when(trainingRepository).selectById("t01");
doReturn(training2).when(trainingRepository).selectById("t02");
```

"t01"を指定してselectByIdメソッドが呼び出されたらtraining1のオブジェクトを返し、  
"t02"を指定してselectByIdメソッドが呼び出されたらtraining2のオブジェクトを返す

```
doReturn(training1, training2).when(trainingRepository).findById(any());
```

1回目にselectByIdが呼び出された時にはtraining1のオブジェクトを返し、  
2回目に呼び出された時にはtraining2のオブジェクトを返す

 doReturn(training1).when(trainingRepository).selectById(any());
doReturn(training2).when(trainingRepository).selectById(any());

上書きする形になるため、selectByIdメソッドを1回目に呼び出した時と2回目  
に呼び出した時で、いずれもtraining2のオブジェクトが返されてしまう

# Mockのメソッドが呼ばれたことをアサーション

```
@Test  
void test_register() {  
    TrainingAdminInput trainingAdminInput = new TrainingAdminInput();  
    trainingAdminInput.setTitle("SQL入門");  
    trainingAdminInput.setReserved(0);  
    trainingAdminInput.setCapacity(8);  
  
    trainingAdminService.register(trainingAdminInput);  ···①  
  
    verify(trainingRepository).insert(any());  ···②  
}
```

- |   |   |
|---|---|
| ① | Serviceオブジェクトのregisterメソッドを呼び出す。registerメソッドの中では、Repositoryオブジェクトのinsertメソッドが呼び出されている想定 |
| ② | Mockitoクラスのverifyメソッドを呼び出す。Mockのメソッドが呼び出されたことをアサーションすることができる。もし呼び出されていなければ、テストが失敗する     |

# Mockのメソッドが呼ばれたことをアサーション

```
verify(trainingRepository, times(2)).insert(any());
```

2回呼び出されることをアサーション

```
verify(trainingRepository, never()).insert(any());
```

1回も呼び出されていないことをアサーション

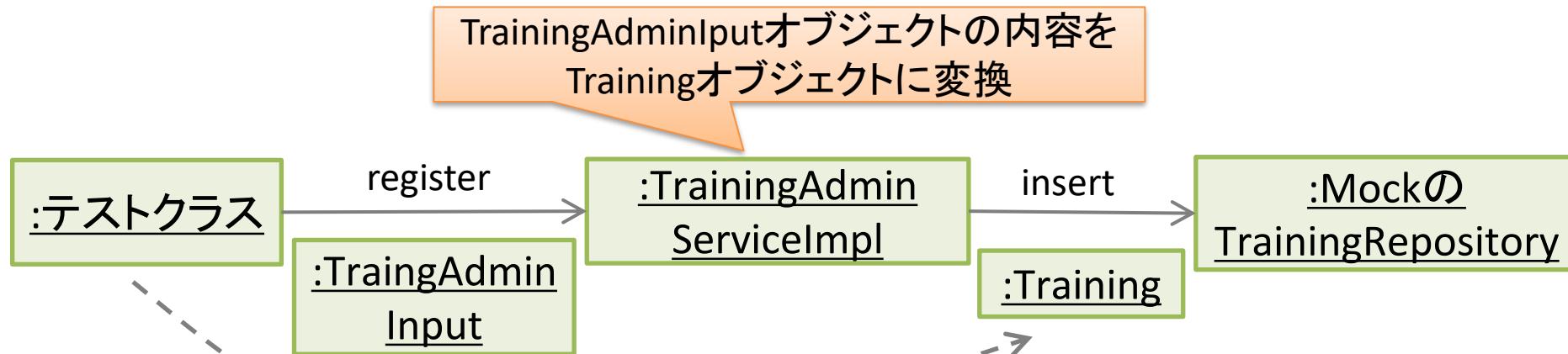
# Mockのメソッドが呼ばれたことをアサーション

- 必要以上にverifyしているケース

```
@Test  
void test_findById() {  
    Training training = new Training();  
    training.setTitle("ビジネスマナー研修");  
    doReturn(training).when(trainingRepository).selectById("t01");  
  
    Training actual = trainingAdminService.findById("t01");  
    assertThat(actual.getTitle()).isEqualTo("ビジネスマナー研修");  
  
    verify(trainingRepository).selectById("t01");  
}
```

このアサーションが成功した時点で、  
selectById("t01")が呼び出されたことは明らか

# Mockのメソッドに渡された引数を確認



TrainingAdminInputオブジェクトの内容が、きちんとTraining  
オブジェクトに反映されているかを確認したい

確認

【registerメソッド】

```
...
@Override
public Training register(TrainingAdminInput trainingAdminInput) {
    Training training = new Training();
    training.setId(UUID.randomUUID().toString());
    training.setTitle(trainingAdminInput.getTitle());
    training.setReserved(trainingAdminInput.getReserved());
    ...
    trainingRepository.insert(training);
    return training;
}
...
```

# Mockのメソッドに渡された引数を確認

```
@Test  
void test_register() {  
    TrainingAdminInput trainingAdminInput = new TrainingAdminInput();  
    trainingAdminInput.setTitle("SQL入門");  
    trainingAdminInput.setReserved(0);  
    trainingAdminInput.setCapacity(8);  
  
    trainingAdminService.register(trainingAdminInput);  ···①  
  
    ArgumentCaptor<Training> trainingCaptor = ArgumentCaptor.forClass(Training.class);  ···②  
    verify(trainingRepository).insert(trainingCaptor.capture());  ···③  
    Training training = trainingCaptor.getValue();  ···④  
    assertThat(training.getTitle()).isEqualTo("SQL入門");  
    assertThat(training.getReserved()).isEqualTo(0);  
    assertThat(training.getCapacity()).isEqualTo(8);  
}
```

①	テスト対象のregisterメソッドを実行
②	ArgumentCaptorクラスのforClassメソッドの引数に、Mockのメソッド(insertメソッド)に渡されるオブジェクトのクラス(Training)を指定
③	verifyでinsertメソッドが呼ばれたことを確認しつつ、①で取得したArgumentCaptorオブジェクトのcaptureメソッドの戻り値をinsertメソッドの引数に指定
④	insertメソッドに渡されたTrainingオブジェクトを取得

# ハンズオン

---

- 「1972-shopping-test-service」

# Mockを使用したテストのデメリット

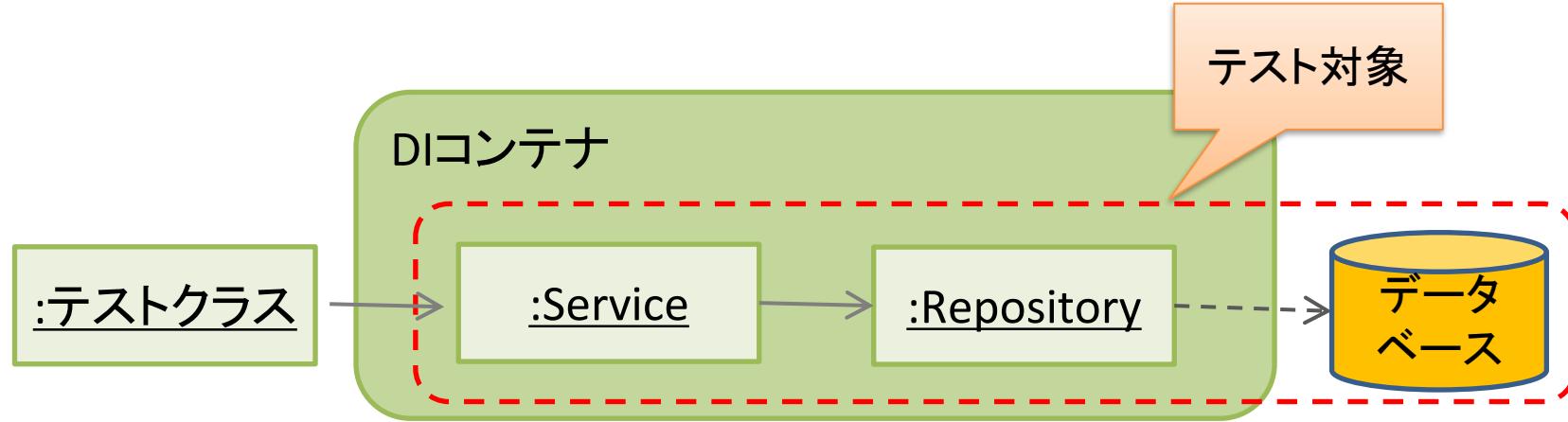
- Serviceの業務ロジックの処理の中で、たくさんの種類のRepositoryオブジェクトのメソッドを呼び出している場合、Mockオブジェクトに挙動を指定したり、Mockオブジェクトのメソッドが適切に呼び出されたことを確認するソースコードが複雑になる
- その状態で、リファクタリングなどでServiceクラスやRepositoryクラスのプログラムを変更した場合、テストクラスの中のMock周りのソースコードが大きく影響を受けることになる
- そのような業務ロジックのテストには、次章で説明する「Service・Repositoryのインテグレーションテスト」の選択肢がある

# **Service・Repositoryのインテグレーションテスト**

# この章の目標

- Service・Repositoryのインテグレーションテストを作成できる
- テストデータの登録とクリーンができる
- 更新系の処理のテストができる

# Service・Repositoryのインテグレーションテストのイメージ



# テスト対象のクラス(プロダクションコード)

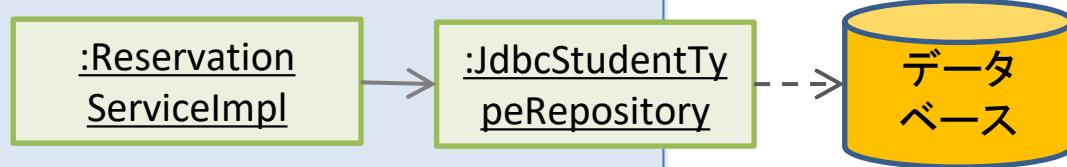
```
@Service  
@Transactional  
public class ReservationServiceImpl implements ReservationService {  
  
    private StudentTypeRepository studentTypeRepository;  
    ...  
  
    public ReservationServiceImpl(StudentTypeRepository studentTypeRepository, ...) {  
        this.studentTypeRepository = studentTypeRepository;  
        ...  
    }  
  
    @Override  
    public StudentType findStudentTypeByCode(String studentTypeCode) {  
        return studentTypeRepository.selectByCode(studentTypeCode);  
    }  
    ...
```

# テストクラスのサンプル

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE) ①
class ReservationServiceTest {

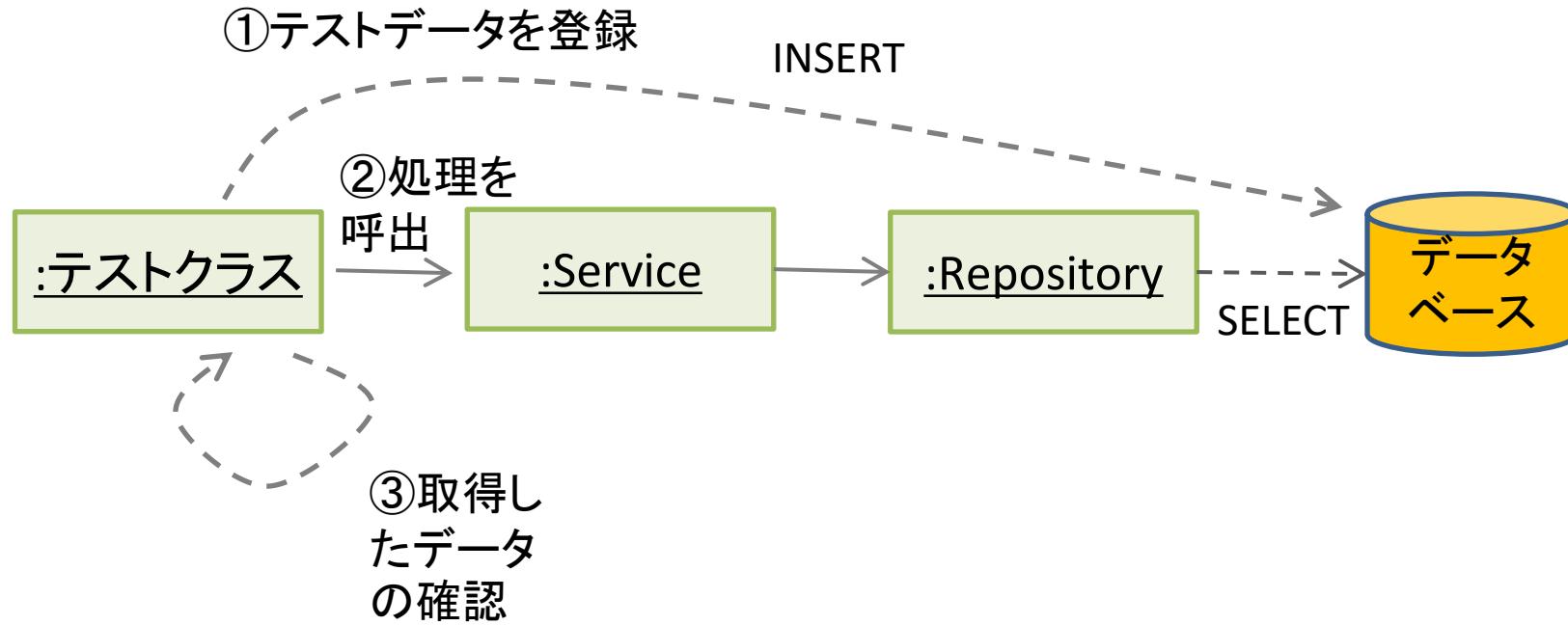
    @Autowired
    ReservationService reservationService; ②

    @Test
    void test_findStudentTypeByCode() {
        StudentType studentType
            = reservationService.findStudentTypeByCode("FREELANCE"); ③
        assertThat(studentType.getName()).isEqualTo("フリーランス"); ④
    }
}
```



- |   |   |
|---|---|
| ① | コンポーネントスキャンやオートコンフィグレーションを制限せずにDIコンテナを生成するアノテーション。「webEnvironment = SpringBootTest.WebEnvironment.NONE」を指定することで、Web周りのコンフィグレーションを無効にしている。読み込むJavaConfigクラスは自動的に探してくれる |
| ② | テスト対象のオブジェクトをインジェクション   |
| ③ | インジェクションしたオブジェクトのメソッドを呼び出す  |
| ④ | 呼び出したメソッドの戻り値が期待通りになっているかをアサーション  |

# データベースのデータの用意



# @Sqlアノテーション

```
@Test  
@Sql("ReservationServiceTest.sql")  
void test_findStudentTypeByCode() {  
    StudentType studentType  
        = reservationService.findStudentTypeByCode("FREELANCE");  
    assertThat(studentType.getName()).isEqualTo("フリーランス");  
}
```

@Sqlはクラスにつけることも可能

# データのクリーン

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@Sql("ReservationServiceTest.sql")
@Transactional ...①
class ReservationServiceTest {
    @Autowired
    ReservationService reservationService;
    @Test
    void test_findStudentTypeByCode() {
        ...
    }
    @Test
    void test_findAllStudentType() {
        ...
    }
}
```

①

テストメソッドの実行と同時にトランザクションが開始され、テストメソッドが終了すると自動的にロールバックされる

# ⟨参考⟩トランザクションの伝搬

## ReservationServiceTest.java

```
@Transactional  
class ReservationServiceTest {  
    ReservationService reservationService;  
    @Test  
    void test() {  
        ...  
        reservationService.reserve(...);  
        ...  
    }  
}
```

トランザクションがコミットされてロールバックされないので？

## ReservationServiceImpl.java

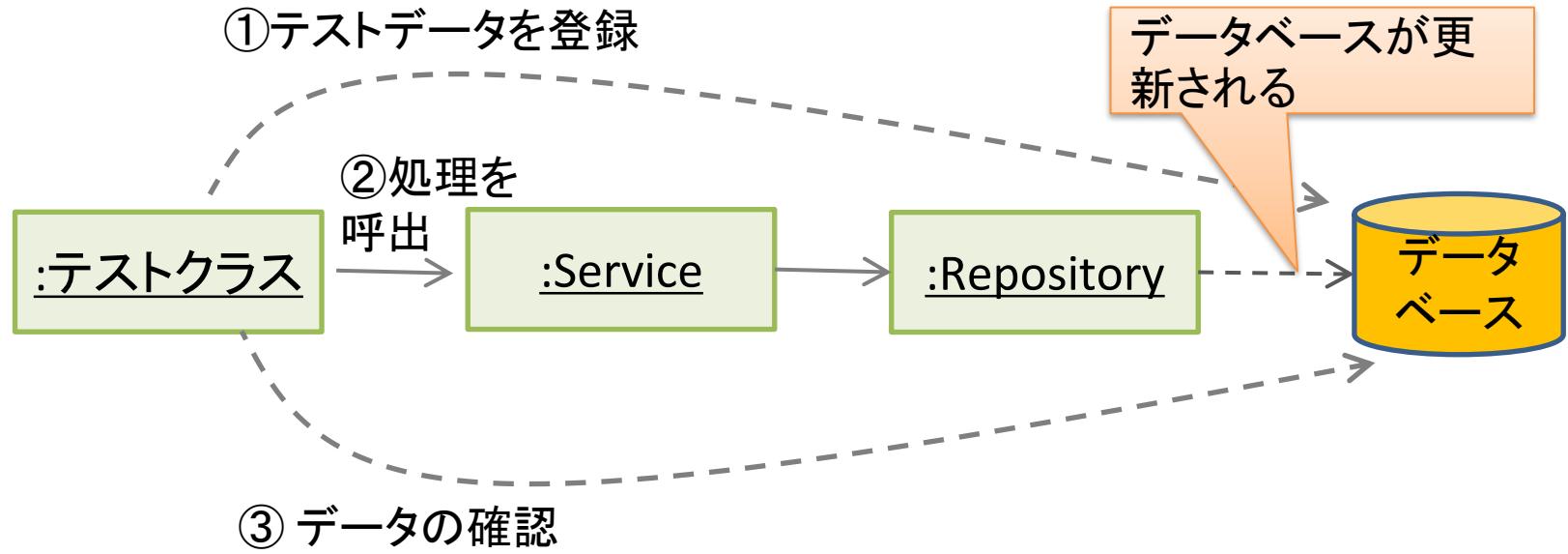
```
@Transactional  
public Reservation reserve(...) {  
    データを更新  
}
```

デフォルトの伝搬の設定「REQUIRED」は、呼び出し側  
すでにトランザクションが開始されていたら、そのト  
ランザクションに加わって処理を行う

トランザクション

テストメソッド終了時にロールバックされるため、  
ReservationServiceImplクラスのメソッドで更新した  
データもロールバックされる

# 更新系の処理のテスト



# JdbcTemplateを使ったデータの確認

```
...
@Autowired
JdbcTemplate jdbcTemplate; ...①

...
@Test
void test_reserve() {
    ...
    Reservation reservation = reservationService.reserve(reservationInput); ...②
    Map<String, Object> reservationMap = jdbcTemplate.queryForMap( ...③
        "SELECT * FROM reservation WHERE id=?", reservation.getId());
    assertThat(reservationMap.get("name")).isEqualTo("東京太郎");
    assertThat(reservationMap.get("phone")).isEqualTo("090-0000-0000");
    ...
    Map<String, Object> trainingMap = jdbcTemplate.queryForMap( ...④
        "SELECT * FROM training WHERE id=?", "t01");
    assertThat(trainingMap.get("reserved")).isEqualTo(4);
}
```

①	JdbcTemplateオブジェクトをインジェクション
②	研修を予約する業務ロジックを呼び出しており、データベースに予約データが登録される
③	登録されたはずの予約レコードをSELECT文で取得し、レコードのカラムの値をアサーション
④	研修テーブルのレコードを取得し、予約数が変わっていることをアサーション

# ハンズオン

---

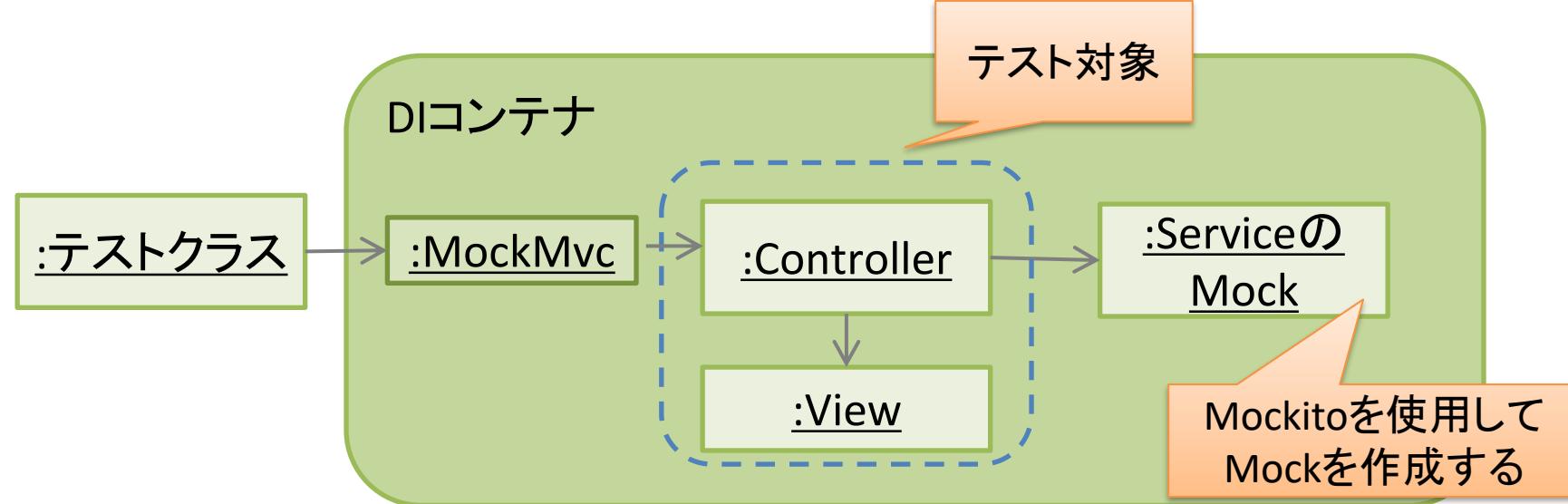
- ・「2002-shopping-test-service-repository」

# **Controllerのユニットテスト**

# この章の目標

- Controllerのユニットテストを作成できる
- MockMvcの基本的な使い方を理解する
- ServiceのMockを作成できる

# Controllerのユニットテストのイメージ



TestClassは、Controllerオブジェクトのメソッドを直接呼び出すのではなく、  
MockMvcを使用して疑似的なリクエストを送信する。ServiceはMockを使用する

# MockMvcとは？

- Controllerに疑似的なHTTPリクエストを送信することができる、Springのテストサポート機能が提供する仕組み
- APサーバを起動しないため、実行時間が速い



# MockMvcの使用イメージ

```
...
@Autowired
MockMvc mockMvc; ...①

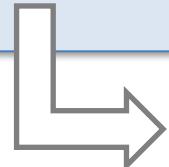
@Test
void test_displayDetails() throws Exception { ...⑥
    ...
    mockMvc.perform( ...②
        MockMvcRequestBuilders.get("/training/display-details")
        .param("trainingId", "t02")
    ) ...③
        .andExpect(MockMvcResultMatchers.status().isOk()) ...④
        .andExpect(MockMvcResultMatchers.view().name("training/displayDetails"))
        .andExpect(MockMvcResultMatchers.content().string(Matchers.containsString("Java研修")))) ...⑤
    ;
}
```

①	MockMvcオブジェクトをテストクラスにインジェクション
②～③	performメソッドでリクエストを送信
④～⑤	レスポンスの内容をアサーション
⑥	performメソッドは、throws句でチェック例外のException型が指定されているためハンドリングが必要。try-catchブロックで囲むのではなく、テストメソッドのthrows句でException型を指定するとよい

# MockMvcとstaticインポート

```
import static org.hamcrest.Matchers.*;  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
```

```
mockMvc.perform(  
    MockMvcRequestBuilders.get("/training/display-details")  
        .param("trainingId", "t02")  
    )  
    .andExpect(MockMvcResultMatchers.status().isOk())  
    .andExpect(MockMvcResultMatchers.view().name("training/displayDetails"))  
    .andExpect(MockMvcResultMatchers.content()  
        .string( Matchers.containsString("Java研修") ))  
;
```



```
mockMvc.perform(  
    get("/training/display-details")  
        .param("trainingId", "t02")  
    )  
    .andExpect(status().isOk())  
    .andExpect(view().name("training/displayDetails"))  
    .andExpect(content()  
        .string(containsString("Java研修") ))  
;
```

# テストクラスのサンプル

## ・ テスト対象のControllerとView

```
@Controller  
public class TrainingController {  
    private TrainingService trainingService;  
    public TrainingController(TrainingService trainingService) {  
        this.trainingService = trainingService;  
    }  
  
    @GetMapping("/training/display-details")  
    public String displayDetails(@RequestParam String trainingId, Model model) {  
        model.addAttribute("training", trainingService.findById(trainingId));  
        return "training/trainingDetails";  
    }  
    ...
```

*training/trainingDetails.html*

```
...  
<h1>研修詳細</h1>  
<table>  
    <tr>  
        <th>研修タイトル</th>  
        <td><span th:text="${training.title}"></span></td>  
    </tr>  
    ...
```

# DIコンテナを生成するためのアノテーションとMock

```
@WebMvcTest(TrainingController.class) ...①
class TrainingControllerTest {

    @Autowired
    MockMvc mockMvc; ...②

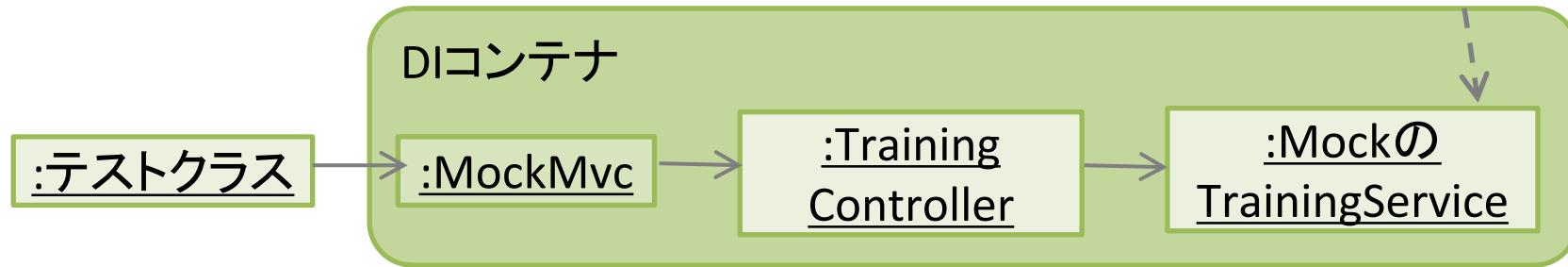
    @MockBean
    TrainingService trainingService; ...③

    @Test
    void test_displayDetails() throws Exception {
    ...
}
```

①	オートコンフィグレーションやコンポーネントスキャンを制限してDIコンテナを生成。() 括弧の中には、テスト対象のControllerクラスを指定。読み込むJavaConfigクラスは自動的に探す
②	MockMvcオブジェクトをインジェクション
③	ServiceのMockオブジェクトを生成。@MockBeanをフィールドに付けた場合は、SpringがMockitoを使ってMockのオブジェクトを生成しフィールドに代入する。さらに、生成したオブジェクトをDIコンテナにBeanとして登録してくれる。@MockBeanはSpringが提供するアノテーション

# Mockのインジェクション

```
@WebMvcTest(TrainingController.class)
class TrainingControllerTest {
    @Autowired
    MockMvc mockMvc;
    @MockBean
    TrainingService trainingService;
    ...
}
```



TrainingControllerオブジェクトにインジェクションされるServiceオブジェクトは、Mockのオブジェクトになる。  
テストメソッドの中で、MockのServiceオブジェクトに挙動を指定できる

# Mockオブジェクトに挙動を指定したサンプル

```
@WebMvcTest(TrainingController.class)
class TrainingControllerTest {
    @Autowired
    MockMvc mockMvc;

    @MockBean
    TrainingService trainingService;

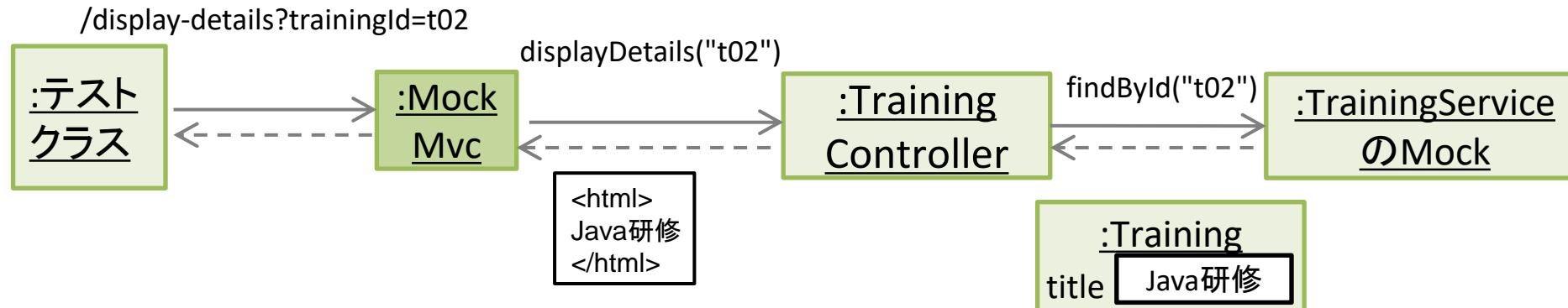
    @Test
    void test_displayDetails() throws Exception {

        Training training = new Training();
        training.setTitle("Java研修");
        doReturn(training).when(trainingService).findById("t02");  ···①

        mockMvc.perform(  ···②
            get("/training/display-details")
            .param("trainingId", "t02")
            ...
            .andExpect(content().string(containsString("Java研修")));  ···③
            ...
    }
}
```

# Mockオブジェクトに挙動を指定したサンプル

- |   |  |
|---|--|
| ① | Mockオブジェクトに挙動を指定。MockのTrainingServiceオブジェクトのfindByIdメソッドが呼ばれたら、タイトルが「Java研修」のTrainingオブジェクトを戻り値で返す挙動を指定  |
| ② | MockMvcオブジェクトを使ってリクエストを送信。リクエストに対応するTrainingControllerオブジェクトのハンドラメソッド(displayDetailメソッドの想定)が呼び出され、ハンドラメソッドの中でMockのTrainingServiceオブジェクトのfindByIdメソッドが呼び出される。findByIdメソッドの戻り値のTrainingオブジェクトがTrainingControllerオブジェクトに返され、タイトルの値がViewオブジェクトによりHTMLの中に埋め込まれる |
| ③ | HTMLの内容をアサーション   |



# リクエストの指定

```
@Test  
void test_displayList() throws Exception {  
    ...  
    mockMvc.perform( ...①  
        get("/training/display-details") ...③  
        .param("trainingId", "t02")  
    ) ...②  
        .andExpect(status().isOk())  
        .andExpect(view().name("training/displayDetails"))  
        .andExpect(content()  
            .string(containsString("Java研修")))  
    ;  
}
```

①～②	performメソッドの呼び出し。performメソッドの引数にリクエストの内容を指定
③	HTTPのGETメソッドのリクエストを指定。引数に指定したURLのパスにリクエストを送信。paramメソッドを呼び出してリクエストパラメータ(パラメータ名「trainingId」、パラメータの値「t02」)を指定

# リクエストの内容を指定する主なメソッド

メソッド名	用途	使用例
get	HTTPのGETメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	get("/display-detail/{id}", "p01")
post	HTTPのPOSTのメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	post("/submit-input")
param	リクエストパラメータを指定する	param("customerName", "東京太郎")
sessionAttr	セッションスコープに任意のオブジェクトを指定する	sessionAttr("customerName", "東京太郎")
flashAttr	フラッシュスコープに任意のオブジェクトを指定する	flashAttr("orderId", "o01")
cookie	Cookieに任意のデータを指定する	cookie("language", "ja")

# レスポンスのアサーション

```
@Test  
void test_displayList() throws Exception {  
    ...  
    mockMvc.perform(  
        get("/training/display-details")  
        .param("trainingId", "t02")  
    )  
        .andExpect(status().isOk()) ...①  
        .andExpect(view().name("training/displayDetails")) ...②  
        .andExpect(content() ...③  
            .string(containsString("Java研修")))) ...④  
    ;  
}
```

①	andExpectメソッドを呼び出して、アサーションしたい内容を引数で渡す。statusメソッドで、ステータスコードが200 OKであることをアサーション
②	viewメソッドを使って、Controllerが戻り値で返したView名をアサーション
③、④	contentメソッドを使ってHTMLの中身をアサーション。HTMLの中に「Java研修」の文字列が含まれていることをチェックしている

# レスポンスの内容をアサーションする主なメソッド

メソッド名	用途	使用例
status	ステータスコード(200 OKなど)を確認する	status().isOk()
view	ハンドラメソッドが返したView名を確認する	view().name("order/form")
content	レスポンスボディの中身を確認する	content().string(containsString("消しゴム"))
request	リクエストスコープやセッションスコープの中身を確認する	request().attribute("userName", "東京太郎")
flash	フラッシュスコープの中身を確認する	flash().attribute("orderId", "o01")
cookie	Cookieのデータを確認する	cookie().value("language", "ja")
model	Modelの中身を確認したり、入力チェックエラーの内容を確認する	model().attribute("userName", "東京太郎")
redirectedUrl	リダイレクトのURLを確認する	redirectedUrl("/order/complete")

# 入力チェックエラーのアサーション

- Spring MVC & Bean Validationの入力チェック

```
public class ReservationInput {  
    private String trainingId;  
    @NotBlank  
    private String name;  
    @NotBlank  
    @Pattern(regexp = "0\d{1,4}-\d{1,4}-\d{4}")  
    private String phone;  
    @NotBlank  
    @Email  
    private String emailAddress;  
    @NotNull  
    private String studentTypeCode;  
    ... Getter・Setterメソッド
```

お名前	<input type="text" value="東京太郎"/>
電話番号	<input type="text" value="abc"/> 電話番号の書式が不正です
メールアドレス	<input type="text" value="xyz"/> メールアドレスの書式が不正です
受講者のタイプ	<input type="button" value="会社員"/>
予約内容を確認	
<a href="#">研修詳細に戻る</a>	

```
@PostMapping("/reservation/validate-input")  
public String validateInput(  
    @Validated ReservationInput reservationInput,  
    BindingResult bindingResult,  
    Model model) {  
    if (bindingResult.hasErrors()) {  
        ...  
        return "reservation/reservationForm";  
    }  
    ...  
    return "reservation/reservationConfirmation";  
}
```

# 入力チェックエラーのアサーション

```
@Test
void test_validateInput_入力エラー() throws Exception {
    mockMvc.perform(
        post("/reservation/validate-input")
    )
    .andExpect(view().name("reservation/reservationForm"))
    .andExpect(model().attributeHasFieldErrorCode("reservationInput", "name", "NotBlank")) ...①
    .andExpect(model().attributeHasFieldErrorCode("reservationInput", "phone", "NotBlank")) ...②
    .andExpect(model().attributeHasFieldErrorCode("reservationInput", "emailAddress", "NotBlank")) ...③
    .andExpect(model().attributeHasFieldErrorCode("reservationInput", "studentTypeCode", "NotBlank")) ...④
    ;
}
```

①～④ modelメソッドの戻り値に対して、attributeHasFieldErrorCodeメソッドを呼び出し、期待するエラーの内容を引数で指定。第1引数はModelオブジェクトの中のInputオブジェクトの名前で、第2引数はプロパティ名、第3引数はエラーコードを指定。エラーコードは、アノテーション名を使用できる

# 入力チェックエラーのアサーション(簡易版)

```
@Test  
void test_validateInput_入力エラー() throws Exception {  
    mockMvc.perform(  
        post("/reservation/validate-input")  
    )  
    .andExpect(view().name("reservation/reservationForm"))  
    .andExpect(model().attributeHasFieldErrors(  ···①  
        "reservationInput", "name", "phone", "emailAddress", "studentTypeCode"))  ···②  
    ;  
}
```

①、②

modelメソッドの戻り値に対してattributeHasFieldErrorsメソッドを呼び出している。  
第1引数にModelオブジェクトの中のInputオブジェクトを指定し、第2引数以降  
は、エラーがあったプロパティ名を指定。指定したプロパティで入力チェックエ  
ラーがあることをアサーション

# デバッグ用のログ出力

```
mockMvc.perform(  
    get("/training/display-details")  
    .param("trainingId", "t02")  
)  
.andExpect(status().isOk())  
.andExpect(view().name("training/displayDetails"))  
.andExpect(content()  
    .string(containsString("Java研修")))  
.andDo(print()) ・・・①  
;
```

- ① andExpectメソッドに繋げてandDoメソッドを呼び出す。  
引数はprintメソッド(MockMvcResultHandlersクラス)の戻り値を渡す

# デバッグ用のログ出力

```
MockHttpServletRequest:  . . . ①
HTTP Method = GET
Request URI = /training/display-details
Parameters = {trainingId=[t02]}
Headers = []
Body = null
Session Attrs = {}
...
MockHttpServletResponse:  . . . ②
Status = 200
Error message = null
Headers = [Content-Language:"en", Content-Type:"text/html;charset=UTF-8"]
...
```

① リクエストの内容が出力される

② レスポンスの内容が出力される

デバッグして問題が解決したら、  
「.andDo(print())」の行は削除する

# ハンズオン

---

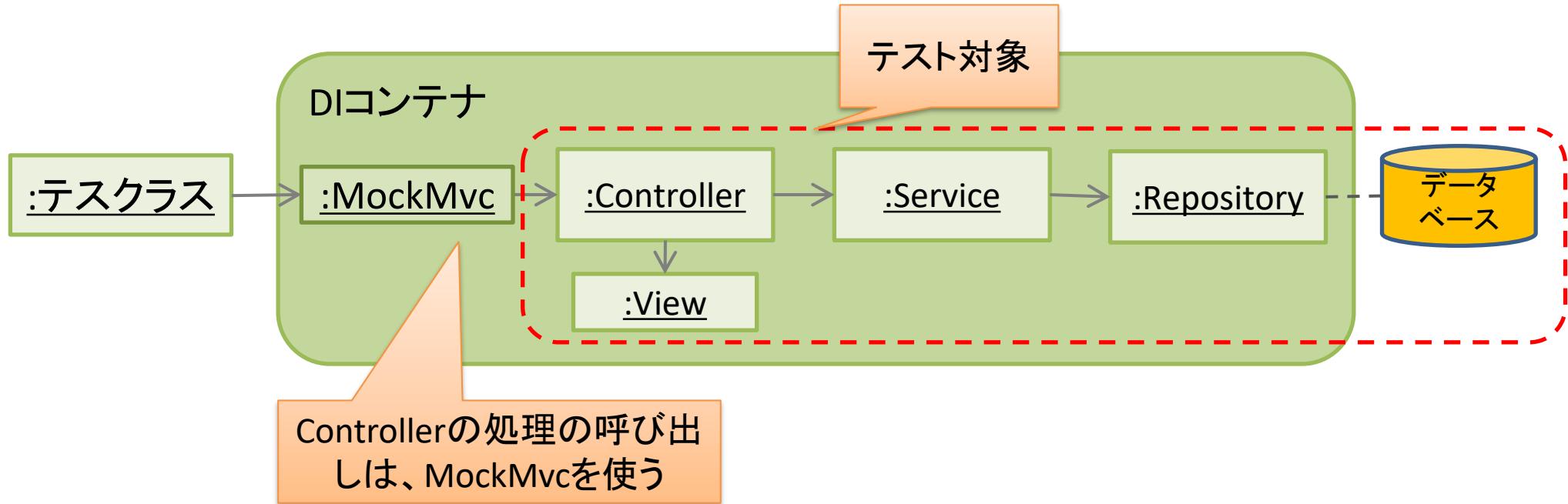
- ・「2102-shopping-test-controller」

# **Controller・Service・Repositoryのインテグレーションテスト**

# この章の目標

- Controller・Service・Repositoryのインテグレーションテストを作成できる

# テストのイメージ



# テストクラスの概要

- @SpringBootTestでDIコンテナを生成
  - オートコンフィグレーションやコンポーネントスキャンを制限しない
- MockMvcを使ってControllerの処理の呼び出しやアサーションを行う
- @Sqlでテストデータを用意
- JdbcTemplateを使って更新後のデータを確認
- トランザクションのロールバックでテストデータをクリーン

# テストクラスのサンプル

```
@SpringBootTest ...①
@AutoConfigureMockMvc ...②
@Transactional ...③
@Sql("TrainingControllerIntegrationTest.sql") ...④
public class TrainingControllerIntegrationTest {
    @Autowired
    private MockMvc mockMvc; ...⑤
    @Test
    public void test_displayDetails() throws Exception {
        mockMvc.perform( ...⑥
            get("/training/display-details")
            .param("trainingId", "t02")
        )
        .andExpect(content().string(containsString("Java実践")))
    }
}
```

①	DIコンテナを生成するためのアノテーション。コンポーネントスキャンやオートコンフィグレーションを制限せずにDIコンテナを生成。JavaConfigクラスを自動的に探す
②	MockMvcオブジェクトをコンフィグレーションするためのアノテーション
③	データベースをロールバックしてデータをクリーンするためのアノテーション
④	テストデータを用意するSQLを実行
⑤	MockMvcオブジェクトをインジェクション
⑥	MockMvcオブジェクトを使ってリクエストを送信

# Modelの中のデータの参照

```
@Test  
public void test_reserve() throws Exception {  
    MvcResult mvcResult = mockMvc.perform(  
        post("/reservation/reserve")  
            .param("reserve", "")  
            .param("name", "東京太郎")  
            .param("phone", "090-0000-0000")  
            ...  
    )  
        .andExpect(view().name("reservation/reservationCompletion"))  
        .andReturn(); ①
```

```
Reservation reservation = (Reservation)mvcResult.getModelAndView()  
    .getModel().get("reservation"); ②  
Map<String, Object> reservationMap = jdbcTemplate.queryForMap(  
    "select * from reservation where id=?", reservation.getId());  
...
```

## テスト対象のController

```
@PostMapping(value = "/reserve", params = "reserve")  
public String reserve(  
    @Validated ReservationInput reservationInput, Model model) {  
    Reservation reservation = reservationService.reserve(reservationInput);  
    model.addAttribute("reservation", reservation);  
    return "reservation/reservationCompletion";  
}
```

発行された予約IDを使って、データベースのレコードをチェックしたい

①	andReturnメソッドでMvcResultオブジェクトを取得
②	MvcResultオブジェクトのメソッドを呼び出してModelオブジェクトを取得し、中のReservationオブジェクトを取得

# ハンズオン

---

- ・「2152-shopping-test-controller-service-repository」

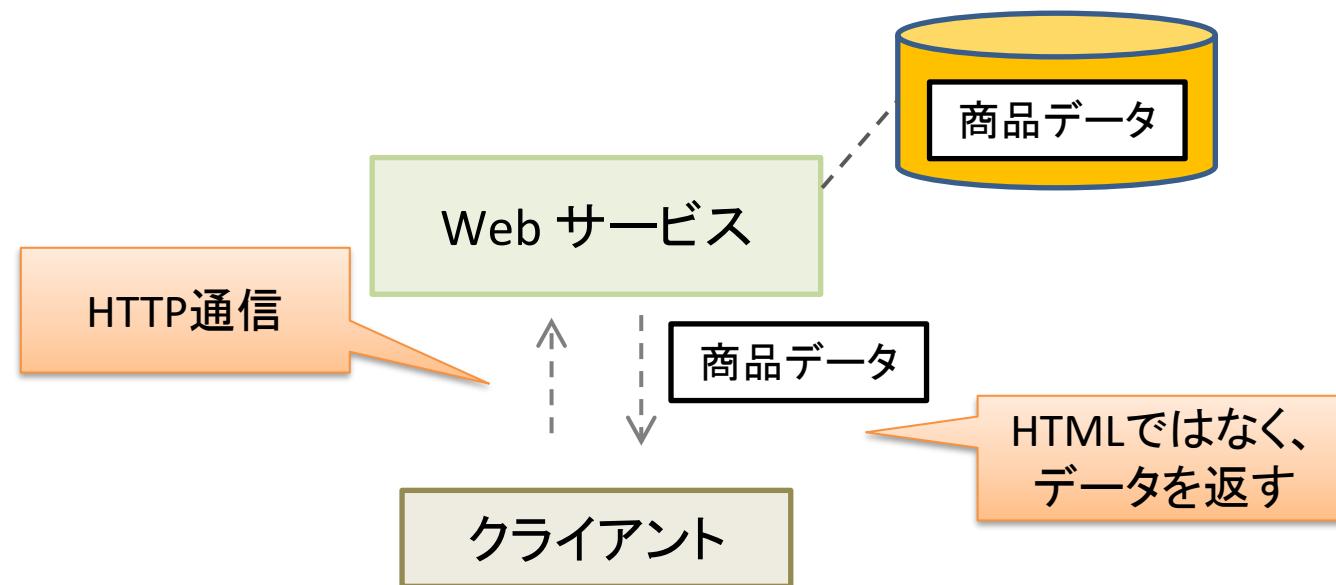
# **RESTful Webサービスのテスト**

# この章の目標

- RESTful Webサービスを、MockMvcを使ってテストできる
- JSONデータの作成やアサーションができる
- RESTful Webサービスを、TestRestTemplateを使ってテストできる

# Webサービスとは？

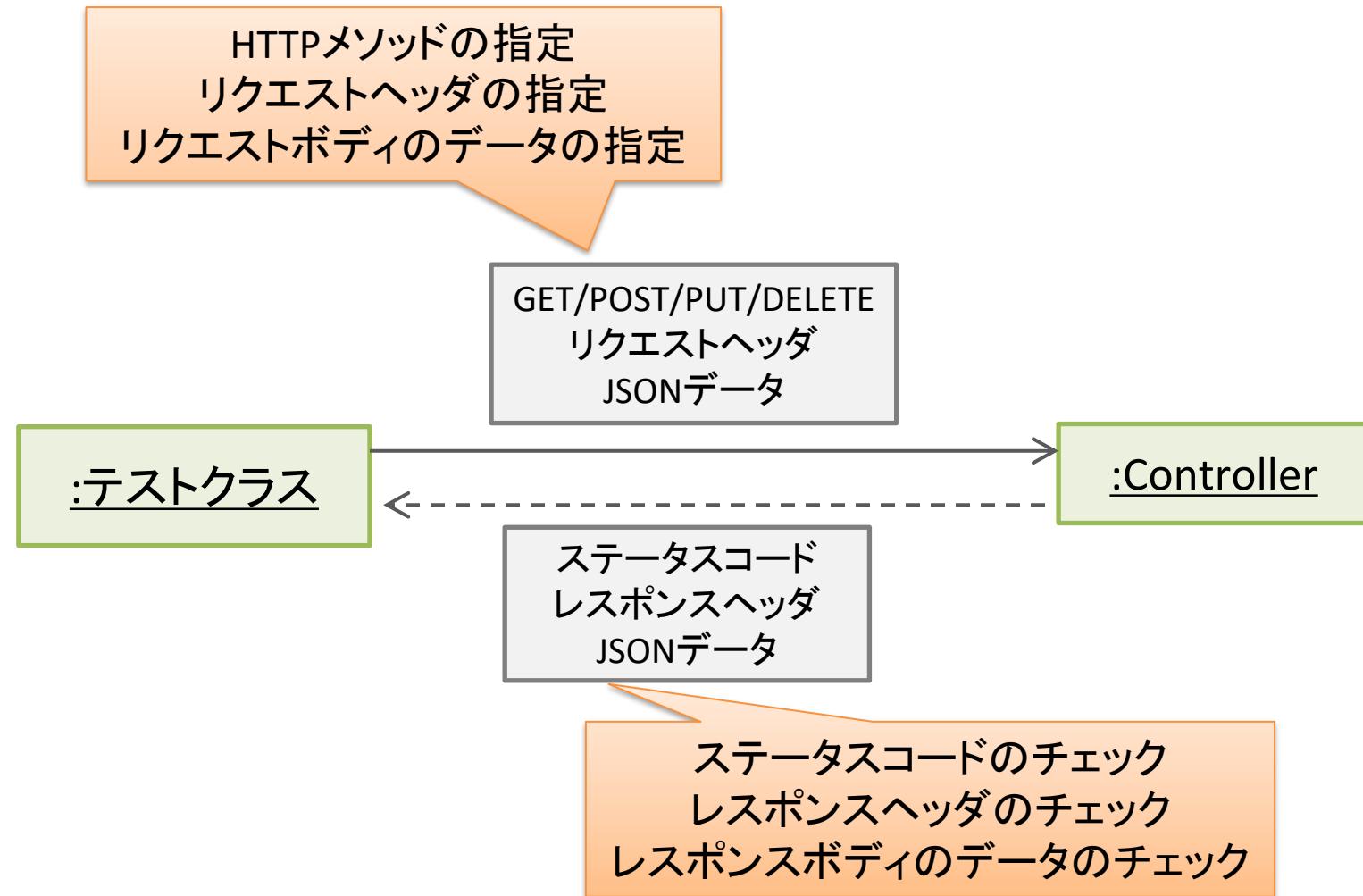
- ・ クライアントへのHTTPレスポンスとして、HTMLではなく、データそのもののを返すアプリケーション



# RESTful Webサービスとは？

- RESTのガイドラインに沿ったWebサービス
- RESTのガイドラインの主な特徴
  - URLでリソースを特定
  - HTTPメソッドでリソースへの操作を指定
  - リクエスト・レスポンスボディのデータ形式をクライアントが指定
  - ステータスコードを細かく使い分ける
  - リクエスト・レスポンスヘッダを活用する

# RESTful Webサービスのテストで必要なこと



※ JSONおよびHTTPのデータ構造について  
補足あり

# RESTful Webサービスのテストの方法

- 大きく2種類ある

- MockMvcを使う方法

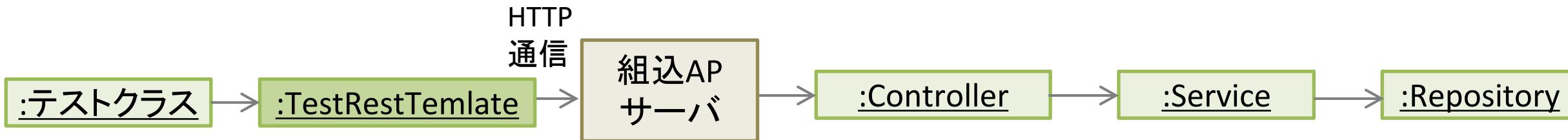
- Controllerのユニットテスト



- Controller・Service・Repositoryのインテグレーションテスト



- TestRestTemplateを使う方法



# MockMvcを使ったControllerのユニットテスト

```
@WebMvcTest(TrainingAdminRestController.class) ...①
class TrainingAdminRestControllerTest {
    @Autowired
    MockMvc mockMvc;
    @MockBean
    TrainingAdminService trainingAdminService;
    @Test
    void test_registerTraining() throws Exception {
        Training training = new Training();
        training.setId("t99");
        doReturn(training).when(trainingAdminService).register(any()); ...②
        String requestBody = """"  ...③
        {
            "title": "SQL入門",
            "startDateTime": "2021-12-01T09:30:00",
            "endDateTime": "2021-12-03T17:00:00",
            "reserved": 0,
            "capacity": 8
        }""";
        mockMvc.perform( ...④
            post("/api/trainings")
            .contentType(MediaType.APPLICATION_JSON) ...⑤
            .content(requestBody) ...⑥
        )
        .andExpect(status().isCreated()) ...⑦
        .andExpect(header().string("Location","http://localhost/api/trainings/t99")) ...⑧
    }
}
```

①	Controllerのユニットテストを行うための @WebMvcTestを付けて、テスト対象のController クラス(TrainingAdminRestControllerクラス)を指定
②	Mockitoを用いて、Serviceオブジェクト (TrainingAdminServiceオブジェクト)のMockが返 す戻り値を指定。新規に登録された研修データ (Trainingオブジェクト)を指定
③	リクエストボディに記述する研修データのJSON データ
④	MockMvcオブジェクトのperformメソッドを呼び出 し、リクエストを送信
⑤	リクエストヘッダの「Content-Type」を指定
⑥	③で用意したJSONデータをリクエストボディに設 定
⑦	ステータスコードの201 Createdが返却されてい ることを確認
⑧	レスポンスヘッダのLocationヘッダの値を確認

# MockMvcを使ったインテグレーションテスト

```
@SpringBootTest ...①
@AutoConfigureMockMvc ...②
@Sql("TrainingAdminControllerIntegrationTest.sql") ...④
@Transactional ...⑤
class TrainingAdminControllerIntegrationTest {
    @Autowired
    MockMvc mockMvc; ...③
    @Test
    void test_registerTraining() throws Exception {
        String requestBody = """
            {
                "title": "SQL入門",
                "startDateTime": "2021-12-01T09:30:00",
                "endDateTime": "2021-12-03T17:00:00",
                "reserved": 0,
                "capacity": 8
            }""";
        mockMvc.perform(
            post("/api/trainings")
                .contentType(MediaType.APPLICATION_JSON)
                .content(requestBody)
        )
        .andExpect(status().isCreated())
        .andExpect(header().string(
            "Location", matchesPattern("http://localhost/api/trainings/.*")))
    ;
}
}
```

①	@SpringBootTestを指定
②	MockMvcオブジェクトをコンフィグレーションしてもらうためのアノテーション
③	MockMvcをインジェクション
④	データベースにデータを登録するSQLファイルを読み込んでもらうアノテーション
⑤	テストメソッド実行後に自動的にデータベースをロールバックしてもらうアノテーション

# リクエストの内容を指定する主なメソッド

メソッド名	用途	使用例
get	HTTPのGETメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	get("/products/{id}", "p01")
post	HTTPのPOSTのメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	post("/products")
put	HTTPのPUTのメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	put("/products/{id}", "p01")
delete	HTTPのDELETEのメソッドを指定する。引数にパスを指定する。パスの中に可変の部分があれば、第2引数以降に値を指定する	delete("/products/{id}", "p01")
contentType	リクエストヘッダの「Content-Type」を指定する	contentType(MediaType.APPLICATION_JSON)
header	任意のリクエストヘッダを指定する	header("CUSTOM_HEADER", "foo")
content	リクエストボディを指定する	content("{\"foo\":\"bar\"}")

# レスポンスの内容をアサーションする主なメソッド

メソッド名	用途	使用例
status	ステータスコード(200 OKなど)を確認する	status().isNoContent()
header	レスポンスヘッダの値を確認する	header().string( "Location","http://localhost/products/p99")
content	レスポンスボディの中身を確認する	content().json("{...}")
jsonPath	JSONPath(後述)を使用してレスポンスボディの中身を確認する	jsonPath("\$.title").value("Java研修")
xpath	XPath(XMLデータ内の要素や属性を特定できる標準的な書式)を使用してレスポンスボディの中身を確認する	xpath("/Product/name").string("消しゴム")

# JSONの文字列を簡単に生成する方法

- JacksonというライブラリのObjectMapperオブジェクトを使用する

```
...
@Autowired
ObjectMapper objectMapper; ...①

@Test
void test_registerTraining() throws Exception {
    ...
    TrainingAdminInput trainingAdminInput = new TrainingAdminInput(); ...②
    trainingAdminInput.setTitle("SQL入門");
    trainingAdminInput.setStartTime(LocalDateTime.of(2021, 12, 1, 9, 30));
    trainingAdminInput.setEndTime(LocalDateTime.of(2021, 12, 3, 17, 0));
    trainingAdminInput.setReserved(0);
    trainingAdminInput.setCapacity(8); ...③
    String requestBody = objectMapper.writeValueAsString(trainingAdminInput); ...④
```

```
{
    "title": "SQL入門",
    "startDateTime": "2021-12-01T09:30:00",
    "endDateTime": "2021-12-03T17:00:00",
    "reserved": 0,
    "capacity": 8
}
```

①	ObjectMapperオブジェクトをインジェクション
②～③	JSONデータの元となるJavaのオブジェクトを生成
④	ObjectMapperオブジェクトのwriteValueAsStringメソッドを使用して、JSONの文字列を生成

# JSONの文字列を簡単に生成する方法

- しかし、前ページの方法だと、TrainingAdminInputクラスの間違いを検知できないという欠点がある
  - 例えば、JSONのメンバで「startDateTime」というメンバを含める必要があった際に、TrainingAdminInputクラスで間違って「startDate」というプロパティ名で実装してしまった場合、生成されるJSONのメンバが「startDate」となってしまうが、そのJSONデータを受け取ったControllerでは、TrainingAdminInputオブジェクトに変換するので問題なく動いてしまう

```
@PostMapping  
public ResponseEntity<Void> registerTraining(@RequestBody TrainingAdminInput trainingAdminInput) {  
...}
```

Controllerでも誤りのあるTrainingAdminInputを使用しているので、動いてしまう

# JSONの文字列を簡単に生成する方法

- Mapを使用した変換

```
...
@Autowired
ObjectMapper objectMapper;

@Test
void test_registerTraining() throws Exception {
    ...
    Map<String, Object> data = new HashMap<>();
    data.put("title", "SQL入門");
    data.put("startDateTime", "2021-12-01T09:30:00");
    data.put("endDateTime", "2021-12-03T17:00:00");
    data.put("reserved", 0);
    data.put("capacity", 8);
    String requestBody = objectMapper.writeValueAsString(data);
    ...
}
```

この方法であれば、誤りのあるInputクラスのオブジェクトに  
変換できないため、誤りに気付くことができる

補足あり

# JSONPathを使用したレスポンスの確認

```
@Test  
void test_getTraining() throws Exception {  
    Training training = new Training(); ①  
    training.setTitle("Java研修");  
    training.setStartTime(LocalDateTime.of(2021, 12, 1, 9, 30));  
    training.setEndTime(LocalDateTime.of(2021, 12, 3, 17, 0));  
    training.setReserved(3);  
    training.setCapacity(10); ②  
    doReturn(training).when(trainingAdminService).findById("t01"); ③  
  
    mockMvc.perform(  
        get("/api/trainings/{id}", "t01") ④  
        .accept(MediaType.APPLICATION_JSON)  
    )  
        .andExpect(status().isOk())  
        .andExpect(jsonPath("$.title").value("Java研修")) ⑤  
        .andExpect(jsonPath("$.startTime").value("2021-12-01T09:30:00"))  
        .andExpect(jsonPath("$.endTime").value("2021-12-03T17:00:00"))  
        .andExpect(jsonPath("$.reserved").value("3"))  
        .andExpect(jsonPath("$.capacity").value("10"))  
};
```

【レスポンスで返されるJSONデータ】

```
{  
    "title": "Java研修",  
    "startTime": "2021-12-01T09:30:00",  
    "endTime": "2021-12-03T17:00:00",  
    "reserved": 3,  
    "capacity": 10  
}
```

①～②	MockのServiceオブジェクトが戻り値で返す研修データを用意
③	用意した研修データを返すようにMockオブジェクトに指示
④	MockMvcオブジェクトを使ってリクエストを送信
⑤	JSONPathを使用してレスポンスボディの中身をアサーション

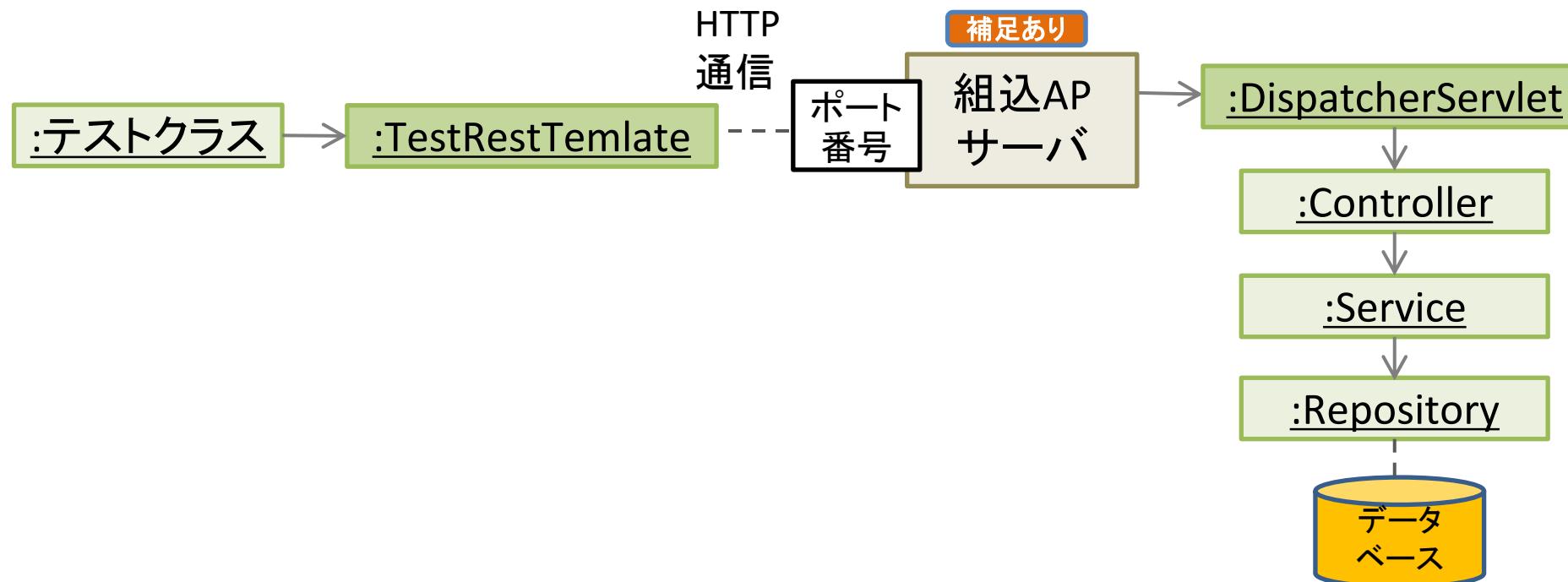
# 「参考」目視でJSONの文字列を確認する方法

```
String responseBody = mockMvc.perform(
    get("/api/trainings/{id}", "t01")
    .accept(MediaType.APPLICATION_JSON)
)
...
.andExpect(jsonPath("$.capacity").value("10"))
.andReturn().getResponse().getContentAsString(StandardCharsets.UTF_8)  ···①
;
// JSONを見やすく整形
String json = objectMapper.readTree(responseBody).toPrettyString();  ···②
System.out.println(json);
```

- ① `andReturn().getResponse()` メソッドを使ってレスポンスのデータを取得し、  
`getContentAsString` メソッドでレスポンスボディのデータを文字列で取得。  
`getContentAsString` メソッドの引数でレスポンスボディのデータの文字コードを指定
  - ② `ObjectMapper` オブジェクトに整形前の JSON の文字列を読み込ませた後、整形後の文  
字列を取得

# TestRestTemplateを使ったテスト

- APサーバを起動して実際にHTTP通信をする
  - より本番に近い形でテストできる
- Controller・Service・Repositoryを繋げたインテグレーションテスト



# TestRestTemplate

- HTTPデータを送受信できるクラス
- RestTemplateをテスト用に拡張したクラス
  - 使い勝手はRestTemplateと同じ
- 4xx、5xxのエラー系のステータスコードが返されたときに例外をスローしないため、テストコードが書きやすい

```
@Test  
void test_getTraining_NotFound() {  
    ResponseEntity<Training> responseEntity  
        = testRestTemplate.getForEntity("/api/trainings/{id}", Training.class, "t99");  
    Assertions.assertThat(responseEntity.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);  
}
```

# テストクラスのサンプル

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT) ・・・①
@Sql("TrainingAdminControllerIntegrationTest.sql") ・・・②
@Sql(value = "clear.sql", executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD) ・・・③
class TrainingAdminControllerIntegrationTestApServer {
    @Autowired
    TestRestTemplate testRestTemplate; ・・・④
    @Test
    void test_getTrainings() {
        ResponseEntity<Training[]> responseEntity = testRestTemplate.getForEntity("/api/trainings", Training[].class);
        assertThat(responseEntity.getStatusCode()).isEqualTo(HttpStatus.OK);
        Training[] trainings = responseEntity.getBody();
        assertThat(trainings.length).isEqualTo(3);
        assertThat(trainings[0].getTitle()).isEqualTo("ビジネスマナー研修");
        assertThat(trainings[1].getTitle()).isEqualTo("Java実践");
    }
}
```

clear.sql

delete from reservation;  
delete from student\_type;  
delete from training;

① webEnvironment属性にWebEnvironment.RANDOM\_PORTを指定。RANDOM\_PORTは、起動するAPサーバのポート番号をランダムに割り振るための設定

② データベースにデータを登録するSQLファイルを読み込んでもらうアノテーション

③ データベースのデータをクリーンするための指定。@Transactionalを付けても、APサーバ上で動くアプリケーションのトランザクションはロールバックされないため、明示的にDELETE文を実行する

④ Springが用意してくれたTestRestTemplateオブジェクトのBeanをインジェクション。TestRestTemplateは、RestTemplateと同じ感覚で使える。APサーバにランダムに割り振られたポート番号が設定済み

# ハンズオン

---

- ・「2251-training-test-rest」

# **Spring Securityのテストサポート**

# この章の目標

- MockMvcと連動したサポート機能を使うことができる
- アノテーションを使用したサポート機能を使うことができる

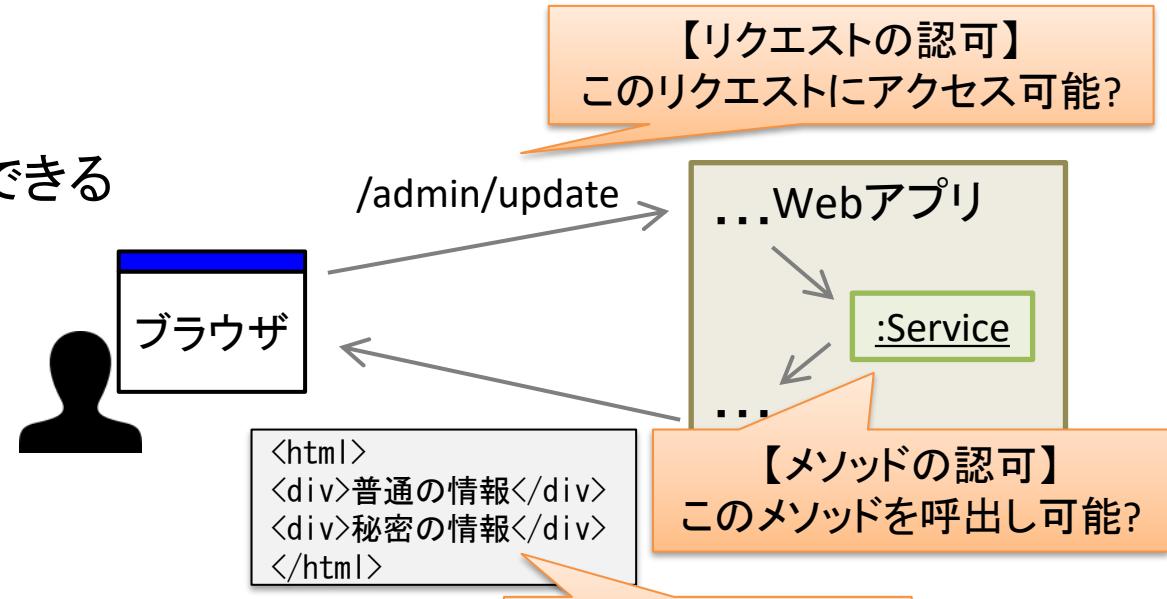
# Spring Securityのテストサポート機能の概要

- MockMvcと連動した機能
  - 認証したユーザの情報を任意に指定できる
    - 認可の挙動を簡単に確認できる
      - リクエストの認可・画面表示の認可の確認に向いている
  - CSRFトークン(後述)をリクエストに含めることができる

```
mockMvc.perform(  
    post("/admin/training/validate-update-input")  
    .with(user("foo").roles("STAFF"))  
    .with(csrf())  
    )...
```

- アノテーションを使用した機能
  - 認証したユーザの情報を任意に指定できる
    - MockMvcを使わない場合の認可の挙動の確認に向いている
      - メソッドの認可の確認時に向いている

```
@Test  
@WithMockUser(roles = "ADMIN")  
void test_delete_ADMINユーザは呼び出せる() {  
    ...
```



# MockMvcと連動した機能の使い方

- MockMvcを使用するテスト
  - Controllerのユニットテスト
  - Controller・Service・Repositoryのインテグレーションテスト

使用するサポート機能はどちらも同じ

※コンフィグレーション(DIコンテナを生成するためのアノテーションなど)は当然異なる

# Controllerのユニットテストのコンフィグレーション

```
...
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*; ...①
...
@WebMvcTest(TrainingAdminController.class)
@Import(SecurityConfig.class) ...②
class TrainingAdminControllerSecurityTest {
    @Autowired
    MockMvc mockMvc; ...③
    ...
}
```

①	SecurityMockMvcRequestPostProcessorsクラスのstaticメソッドをstaticインポート
②	Spring Securityのコンフィグレーションを行っているJavaConfigクラスのSecurityConfigクラスをインポート。 <code>@WebMvcTest</code> がコンポーネントスキャンを制限しており、 <code>@Configuration</code> を、コンポーネントスキャンで取り込まないため、明示的にインポートする必要がある
③	MockMvcオブジェクトをインジェクション

# インテグレーションテストのコンフィグレーション

```
...
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
...
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@Sql("TrainingAdminControllerIntegrationTest.sql")
class TrainingAdminControllerIntegrationTest {
    @Autowired
    MockMvc mockMvc;
    ...
}
```

# 認証したユーザの情報を任意に指定

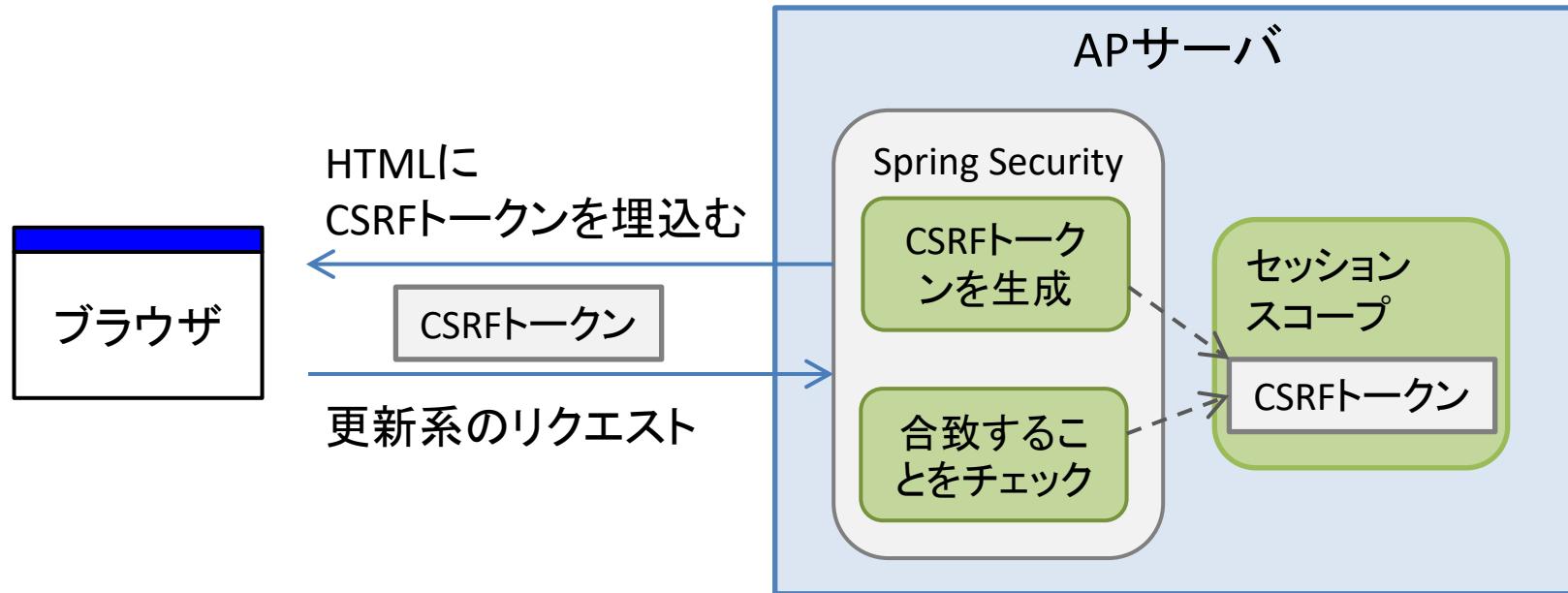
```
import static  
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;  
...  
@Test  
void test_displayList_GUESTユーザはアクセスできない() throws Exception {  
    mockMvc.perform(  
        get("/admin/training/display-list")  
        .with(user("foo").roles("GUEST")) ...①  
    )  
    .andExpect(status().isForbidden()) ...②  
};  
...  
}
```

① withメソッドを使って認証したユーザの情報を指定。  
userメソッドは、SecurityMockMvcRequestPostProcessorsのstaticメソッド

② 403 Forbiddenのステータスコードが返されることをアサーション

# CSRFトークンの指定

- Spring SecurityによるCSRFの対応



POSTのような更新系のリクエストをテストする場合は、CSRFトークンの値をリクエストに含める必要がある。そうしないとエラーが発生してテストできない。  
参照系のGETのリクエストの場合はチェックしないため(意図せずリクエストしたとしても、自分のデータが表示されるだけなので実害がないため)、CSRFトークンの値を含める必要はない

# CSRFトークンの指定方法

```
import static  
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;  
...  
@Test  
void test_validateUpdateInput_ADMINユーザはアクセスできる() throws Exception {  
    mockMvc.perform(  
        post("/admin/training/validate-update-input")  
        .with(user("foo").roles("ADMIN"))  
        .with(csrf())  ...①  
    )  
    .andExpect(status().isOk())  
    ;  
}
```

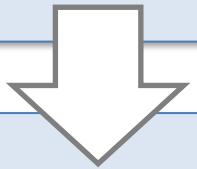
①

withメソッドの引数にcsrfメソッドの戻り値を指定。userメソッドは、SecurityMockMvcRequestPostProcessorsのstaticメソッド。開発者が明示的にCSRFトークンの値を指定しなくても、正しいCSRFトークンの値が自動的にリクエストに含まれる

# アノテーションを使用した機能の使い方

- アノテーションを使用して認証したユーザの情報を任意に指定してテストを実行できる

```
@Test  
void test_displayList_GUESTユーザはアクセスできない() throws Exception {  
    mockMvc.perform(  
        get("/admin/training/display-list")  
        .with(user("foo").roles("GUEST"))  
    )  
    .andExpect(status().isForbidden())  
    ;  
}
```



```
@Test  
@WithMockUser(roles = "GUEST")  
void test_displayList_GUESTユーザはアクセスできない() throws Exception {  
    mockMvc.perform(  
        get("/admin/training/display-list")  
    )  
    .andExpect(status().isForbidden())  
    ;  
}
```

## アノテーションを使用した機能の使いどころ

- MockMvcを使う場合は、MockMvcと連動した機能でこと足りる(アノテーションを使用した機能は使う必要はない)
- MockMvcを使用しないテストで、認証したユーザの情報を指定したい場合に有効
  - 代表的なものとして、メソッドの認可のテストがある

# メソッドの認可のテスト

- ・ メソッドの認可のサンプル

```
@Service  
@Transactional  
public class TrainingAdminServiceImpl implements TrainingAdminService {  
    ...  
    @Override  
    @PreAuthorize("hasRole('ADMIN')") ①···  
    public void delete(String trainingId) {  
        trainingRepository.delete(trainingId);  
    }  
    ...
```

①

@PreAuthorizeアノテーションを付けて、()括弧の中に認可の条件を指定。  
ADMIN権限を持っていないユーザがdeleteメソッドを呼び出そうとすると、  
AccessDeniedExceptionオブジェクトが例外としてスローされる

メソッドの認可は、Serviceのメソッド(業務ロジックのメソッド)で行うのが一般的

# メソッドの認可のテスト(Service・Repositoryのインテグレーションテスト)

```
@SpringBootTest ...①
@Transactional
class TrainingAdminServiceSecurityTest {
    @Autowired
    TrainingAdminService trainingAdminService;
    @Test
    @WithMockUser(roles = "GUEST") ...②
    void test_delete_GUESTユーザは呼び出せない() {
        assertThatThrownBy(() -> {
            trainingAdminService.delete("t01");
        }).isInstanceOf(AccessDeniedException.class); ...④
    }
    @Test
    @WithMockUser(roles = "ADMIN") ...⑤
    void test_delete_ADMINユーザは呼び出せる() {
        trainingAdminService.delete("t01");
    }
    @Test
    @WithAnonymousUser ...⑥
    void test_delete_認証してない場合は呼出せない() {
        assertThatThrownBy(() -> {
            trainingAdminService.delete("t01");
        }).isInstanceOf(AccessDeniedException.class);
    }
}
```

- |   |  |
|---|--|
| ① | @SpringBootTestを指定して、DIコンテナを生成。Web周りのコンフィグレーションが不要なためwebEnvironment属性にWebEnvironment.NONEを指定したいが、Spring Securityを使用する場合は都合が悪い(後述)ため指定していない |
| ② | @WithMockUserを付けて、認証したユーザの情報を指定  |
| ③ | assertThatThrownByメソッド(AssertJが提供するメソッド)を使用し、例外が投げられる想定の処理(deleteメソッドの呼び出し)をラムダ式で渡している   |
| ④ | メソッドチェーンを使ってisInstanceOfメソッドを呼び出し、AccessDeniedExceptionオブジェクトがスローされることをアサーション  |
| ⑤ | ADMIN権限のユーザでdeleteメソッドを呼び出し、問題なく呼び出せることを確認   |
| ⑥ | テストメソッドに@WithAnonymousUserを付けて、認証していないユーザがdeleteメソッドを呼び出せないことを確認   |

# WebEnvironment.NONEを指定しなかった理由

- Spring Securityのリクエストの認可の設定でrequestMatchersメソッドを使用した場合、Web周りのコンフィグレーションがされていないと、DIコンテナ生成時にエラーが発生する

```
@Configuration  
@EnableWebSecurity  
@EnableMethodSecurity  
public class SecurityConfig {  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
        http  
            .authorizeHttpRequests(authorize -> authorize  
                .requestMatchers(HttpMethod.POST, "/admin/**").hasRole("ADMIN")  
                .requestMatchers("/admin/**").hasAnyRole("ADMIN", "STAFF")  
                .anyRequest().permitAll()  
            );  
        return http.build();  
    }  
}
```

Web周りのコンフィグレーションが必要

この理由から、WebEnvironment.NONEを指定せずに、Web周りのコンフィグレーションを有効にしていた

# WebEnvironment.NONEを指定しなかった理由

- 一方、ServiceとRepositoryのインテグレーションテストで純粋に業務ロジックをテストする際は、メソッドの認可の処理が動いてしまうとテストがし難くなるので、Spring Security周りのJavaConfig(前ページのSecurityConfigクラス)は読み込みたくない
- 対応策として、Spring Securityのコンフィグレーションを行っているJavaConfigクラスに @ConditionalOnWebApplicationを付けて、業務ロジックを純粋にテストするServiceと Repositoryのインテグレーションテストでは、@SpringBootTest(webEnvironment = WebEnvironment.NONE)を指定する方法がある

## JavaConfigクラス

```
@ConditionalOnWebApplication  
@Configuration  
@EnableWebSecurity  
@EnableMethodSecurity  
public class SecurityConfig {  
    ...
```

Web周りのコンフィグレーションが有効なときだけSecurityConfigクラスが取り込まれる

## 業務ロジックを純粋にテスト

```
@SpringBootTest(webEnvironment=NONE)  
@Transactional  
class TrainingAdminServiceTest {  
    ...
```

Web周りのコンフィグレーションが無効なので、SecurityConfigクラスは読み込まれない

## メソッドの認可をテスト

```
@SpringBootTest  
@Transactional  
class TrainingAdminServiceSecurityTest {  
    ...
```

Web周りのコンフィグレーションが有効なので、SecurityConfigクラスが読み込まれる

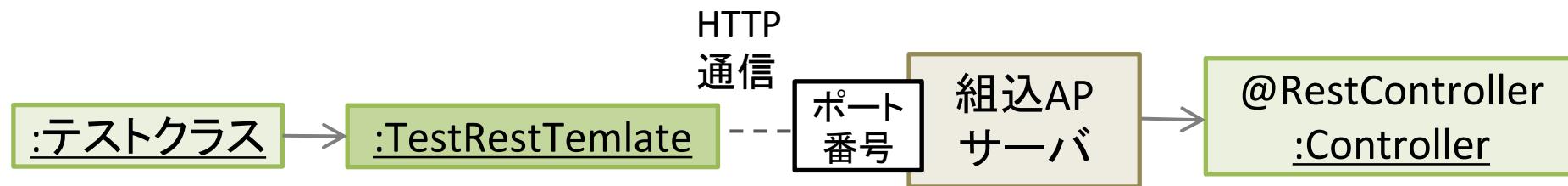
# ハンズオン

---

- ・「2201-training-test-security」

# TestRestTemplateを使用する場合

- MockMvcの仕組みを使う訳ではないため、MockMvcと連動したSpring Securityのテストサポート機能が使えない
- また、@WithMockUserなどのユーザ情報を指定するアノテーションは、APサーバ上で動くアプリケーションには機能しないため使えない
- 認証するための情報(ID・パスワードなど)をリクエストに含める形になる



# TestRestTemplateを使用する場合

- REST APIがBASIC認証に対応している場合

```
@Autowired  
TestRestTemplate testRestTemplate;  
  
@Test  
void test_getTrainings() {  
    ResponseEntity<Training[]> responseEntity = testRestTemplate  
        .withBasicAuth("taro", "taro123") ...①  
        .getForEntity("/api/trainings", Training[].class);  
    ...  
}
```

①

withBasicAuthメソッドを呼び出し、第1引数にユーザ名で、第2引数にパスワードを指定。自動的にAuthorizationヘッダが設定される。withBasicAuthメソッドの戻り値はTestRestTemplateオブジェクトなので、続けてgetForEntityメソッドでGETのリクエストを送信

# 付録

Selenideを用いたE2Eテスト  
フラッシュスコープの扱い  
セッションスコープのBeanの扱い

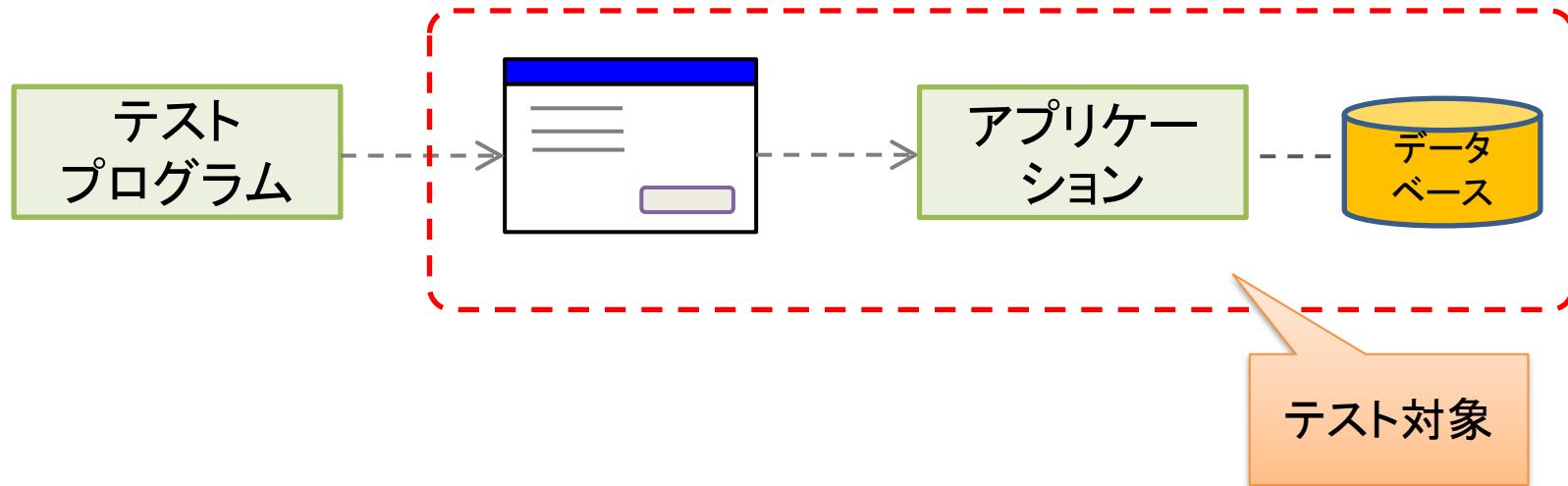
# Selenideを用いたE2Eテスト

# この章の目標

- Selenideの基本的な使い方を理解する
- Selenideを使ってE2Eテストを作成できる

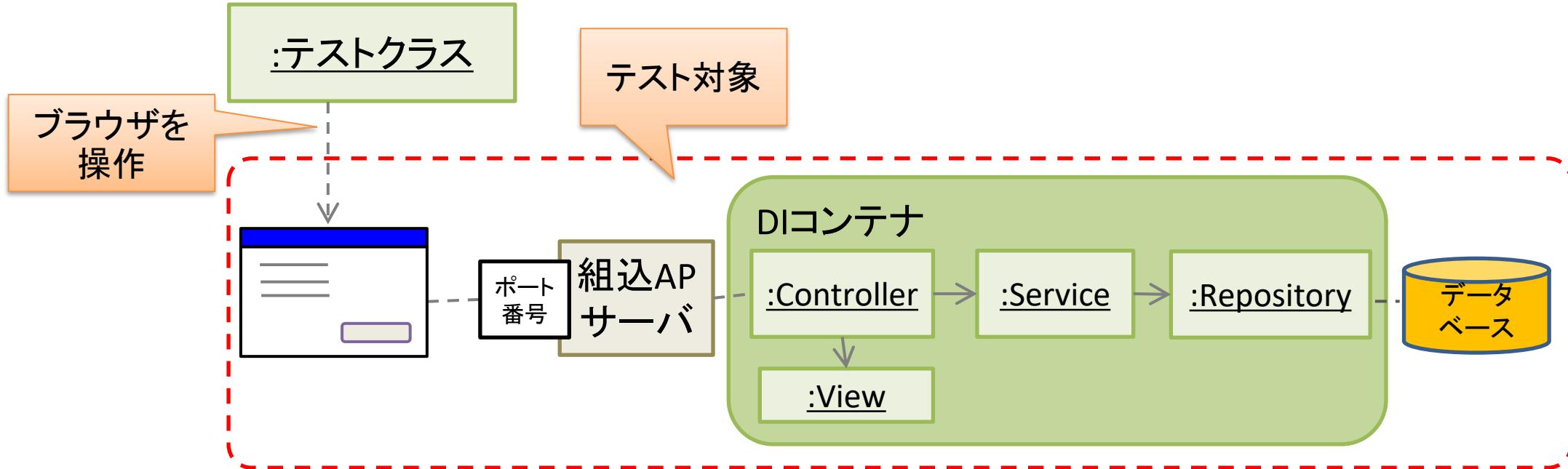
# E2Eテストとは？

- E2Eテスト(エンド・ツー・エンドテスト)
  - アプリケーション全体(端から端まで)を繋げてテストする
  - 画面、アプリケーション、データベースを繋げてテスト



- 画面の操作や挙動の確認は、テストプログラムが行う

# テストのイメージ



# Selenideとは？

- ・ ブラウザを自動的に操作してWebアプリケーションをテストするためのJavaのライブラリ
    - 内部では、ブラウザの操作を自動化するSelenium WebDriverというライブラリが使用されている
    - Selenium WebDriverをテスト向けに拡張している

```
class SampleTest {  
    @Test  
    void test() {  
        Selenide.open("https://www.google.com");  ···①  
        Selenide.$("*[name=q]").should(Condition.focused);  ···②  
    }  
}
```

①	URLを指定してブラウザを起動
②	表示された画面の中の、name属性が「q」のタグ(検索キーワードの入力項目)に対して、フォーカスが当たっていることをアサーション

# @SpringBootTestとの併用

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT) ...①
class ReservationUiTest {
    @Value("${local.server.port}") ...②
    int randomPort;
    @BeforeEach
    void setup() {
        Configuration.baseUrl = "http://localhost:" + randomPort; ...③
    }
    @Test
    void test() {
        Selenide.open("/training/display-list"); ...④
    }
}
```

①	テスト実行時に自動的にDIコンテナが生成される。webEnvironment属性に「WebEnvironment.RANDOM_PORT」が指定されているので、APサーバがランダムなポート番号で起動する
②	DIコンテナが管理するプロパティの「local.server.port」を@Value([chap-external]で紹介)から、起動しているAPサーバが使用しているポート番号を取得
③	Selenideに対して、ポート番号を含めたURLの先頭部分を指定。Configurationクラスは、Selenideのクラス。テストメソッドの中でURLを指定する場合は、ポート番号より後のパスの部分だけ指定すればよい
④	Selenideのopenメソッドを呼び出し、パスを指定してブラウザを起動

# @Sqlとの併用

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class ReservationUiTest {
    @Value("${local.server.port}")
    int randomPort;
    @BeforeEach
    void setup() {
        Configuration.baseUrl = "http://localhost:" + randomPort;
    }
    @Test
    @Sql("ReservationUiTest.sql")  ···①
    void test() {
        Selenide.open("/training/display-list");
        ...
    }
}
```

- ① テストメソッドが実行されるタイミングで、指定したSQLファイルが自動的に読み込まれ実行される

# データのクリーン

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class ReservationUiTest {
    ...
    @Test
    @Sql("ReservationUiTest.sql")
    @Sql(value = "clear.sql",  ...①
          executionPhase = ExecutionPhase.AFTER_TEST_METHOD)  ...②
    void test() {
        Selenide.open("/training/display-list");
        ...
    }
}
```

clear.sql

```
delete from reservation;
delete from student_type;
delete from training;
```

- |   |   |
|---|---|
| ① | APサーバを起動してテストする場合は、@Transactionalを付けてもロールバックされないため、明示的にDELETE文を発行する |
| ② | テストメソッドの実行後にSQLファイルを実行する  |

@Sqlはクラスの上に付けることも可能

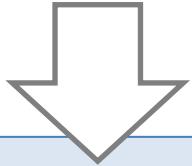
# Selenideの使い方

- staticインポート
- ブラウザの起動
- 画面要素の参照
- 画面要素に対する操作
- 画面要素の情報を取得
- 画面要素に対するアサーション

# Selenideとstaticインポート

```
import static com.codeborne.selenide.Selenide.*;
import static com.codeborne.selenide.Condition.*;
import static com.codeborne.selenide.Selectors.*;
```

```
@Test
void test() {
    Selenide.open("https://www.google.com");
    Selenide.$("*[name=q]").should(Condition.focused);
}
```



```
@Test
void test() {
    open("https://www.google.com");
    $("*[name=q]").should(focused);
}
```

# ブラウザの起動

- openメソッド
  - open(URLの文字列)
  - Selenideによって既にブラウザが起動済みの場合は、起動済みのブラウザに対して指定したURLの画面が読み込まれる

```
@Test  
void test() {  
    open("/training/display-list");  
    ...  
}
```

# 画面要素の参照

- \$メソッド
  - \$(要素の検索条件)
    - 検索条件に合致した最初の要素を参照
  - \$\$ (要素の検索条件)
    - 検索条件に合致したすべての要素を参照

# 画面要素の検索条件の指定

- Byオブジェクトを使用する方法

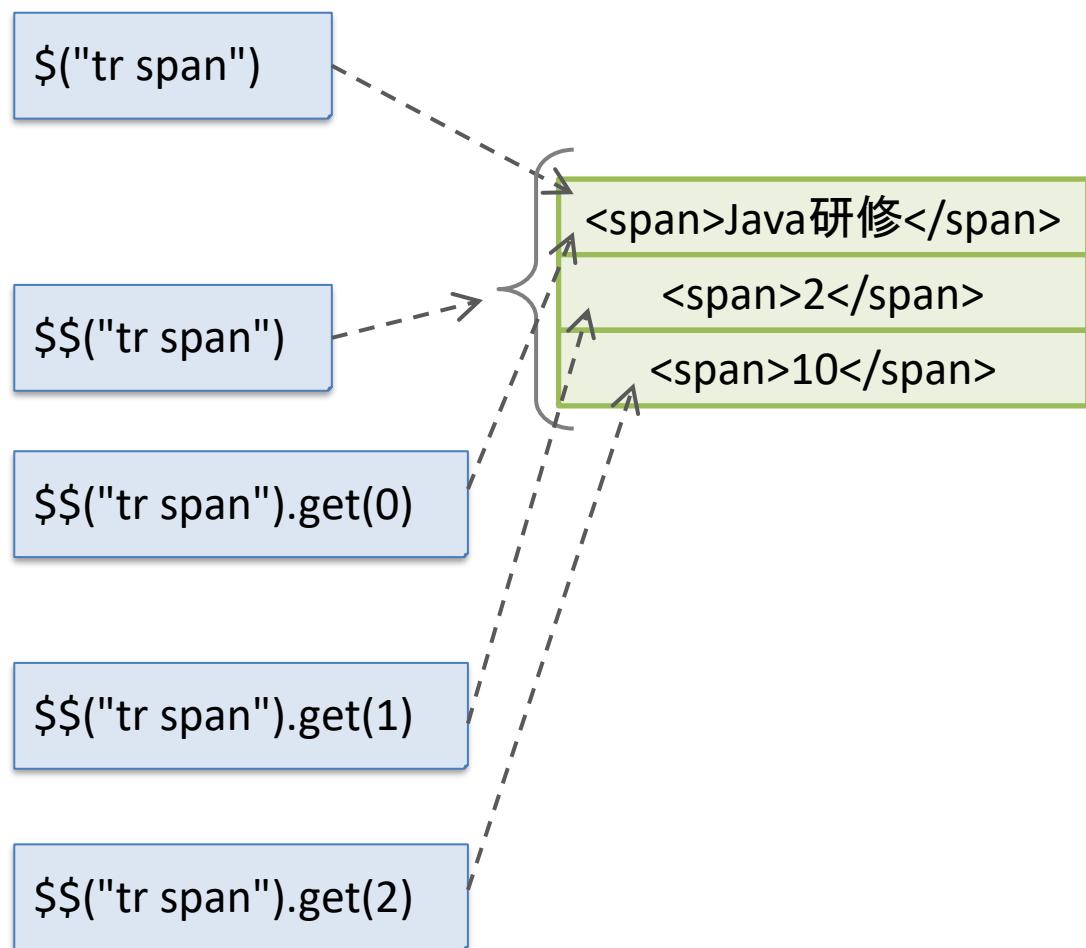
```
$(byTagName("title")) // タグ名を指定して検索  
$(byClassName("error")) // クラス名を指定して検索
```

- CSSセレクターを使用する方法

```
$("#title") // タグ名を指定して検索  
$(".error") // クラス名を指定して検索  
$("a[href*=trainingId]") // アンカータグのhref属性の値に「trainingId」の文字が含まれる要素を検索
```

本研修ではCSSセレクターを使用

# \$と\$\$の違い



```
<table>
  <tr>
    <th>研修タイトル</th>
    <td>
      <span>Java研修</span>
    </td>
  </tr>
  <tr>
    <th>予約数</th>
    <td>
      <span>2</span>
    </td>
  </tr>
  <tr>
    <th>定員</th>
    <td>
      <span>10</span>
    </td>
  </tr>
</table>
```

# 画面要素に対する操作

- 操作を行うための代表的なメソッド

メソッド	用途	使用例
click()	クリックの操作を行う	\$("#some-button").click()
selectRadio(選択肢の値)	選択肢の値(value属性の値)を指定してラジオボタンを選択する	\$("input[name=some-radio]").selectRadio("JAPAN")
selectOptionByValue(選択肢の値)	選択肢の値(value属性の値)を指定してプルダウンを選択する	\$("select[name=some-select]").selectOptionByValue("JAPAN")
setValue(文字列)	指定した文字列を入力フィールドに設定する	\$("#some-input").setValue("東京太郎")

```
$( "input[name=phone]" ).setValue("090-0000-0000")
```

name属性の値が「phone」のinput要素に  
「090-0000-0000」を入力

# 要素の情報を取得

- 情報を取り得するための代表的なメソッド

メソッド	用途	使用例
text()	要素の中のテキストを取得する	\$("#some-span").text()
val()	value属性の値を取得する	\$("#some-input").val()
attr(属性名)	指定した属性の値を取得する	\$("#some-div").attr("class")
getSelectedOptionValue()	プルダウンで選択された値を取得する	\$("select[name=some-select]").getSelectedOptionValue()

```
$( "tr span" ).text()
```

```
<tr>
  <th>研修タイトル</th>
  <td><span>Java研修</span></td>
</tr>
```

「Java研修」の文字列が取得される

# 画面要素に対するアサーション

- 参照した要素のshouldメソッドを呼び出して、引数にアサーションしたい内容を指定
  - \$(要素の検索条件).should(アサーションの内容)
  - shouldBe、shouldHaveメソッドもあるが、いずれもshouldの別名なので挙動は同じ
  - アサーションの内容は、Conditionオブジェクトを使用する。Conditionオブジェクトは、Conditionクラスのstaticメソッドやstaticフィールドで取得できる

# 画面要素に対するアサーション

- Conditionクラスの主要なstaticメソッドおよびstaticフィールド

メソッド・フィールド	用途	使用例
attribute(属性名, 値)	指定した属性が、期待する値になっていることを確認する	<code>\$("#some-div").should(attribute("class", "error"))</code>
exactValue(値)	value属性の値が、期待する値になっていることを確認する	<code>\$("#some-input").should(exactValue("東京太郎"))</code>
text(値)	要素の中のテキストが、期待する値を含んでいることを確認する	<code>\$("#some-span").should(text("Java"))</code>
exactText(値)	要素の中のテキストが、期待する値になっていることを確認する	<code>\$("#some-span").should(exactText("Java研修"))</code>
matchText(正規表現)	要素の中のテキストが、期待する値になっていることを正規表現で確認する	<code>\$("#some-span").should(text("ビジネス.*研修"))</code>
focused	要素にフォーカスが当たっていることを確認する	<code>\$("#some-button").should(focused)</code>
disabled	要素が無効になっていることを確認する	<code>\$("#some-button").should(disabled)</code>

# 画面要素に対するアサーション

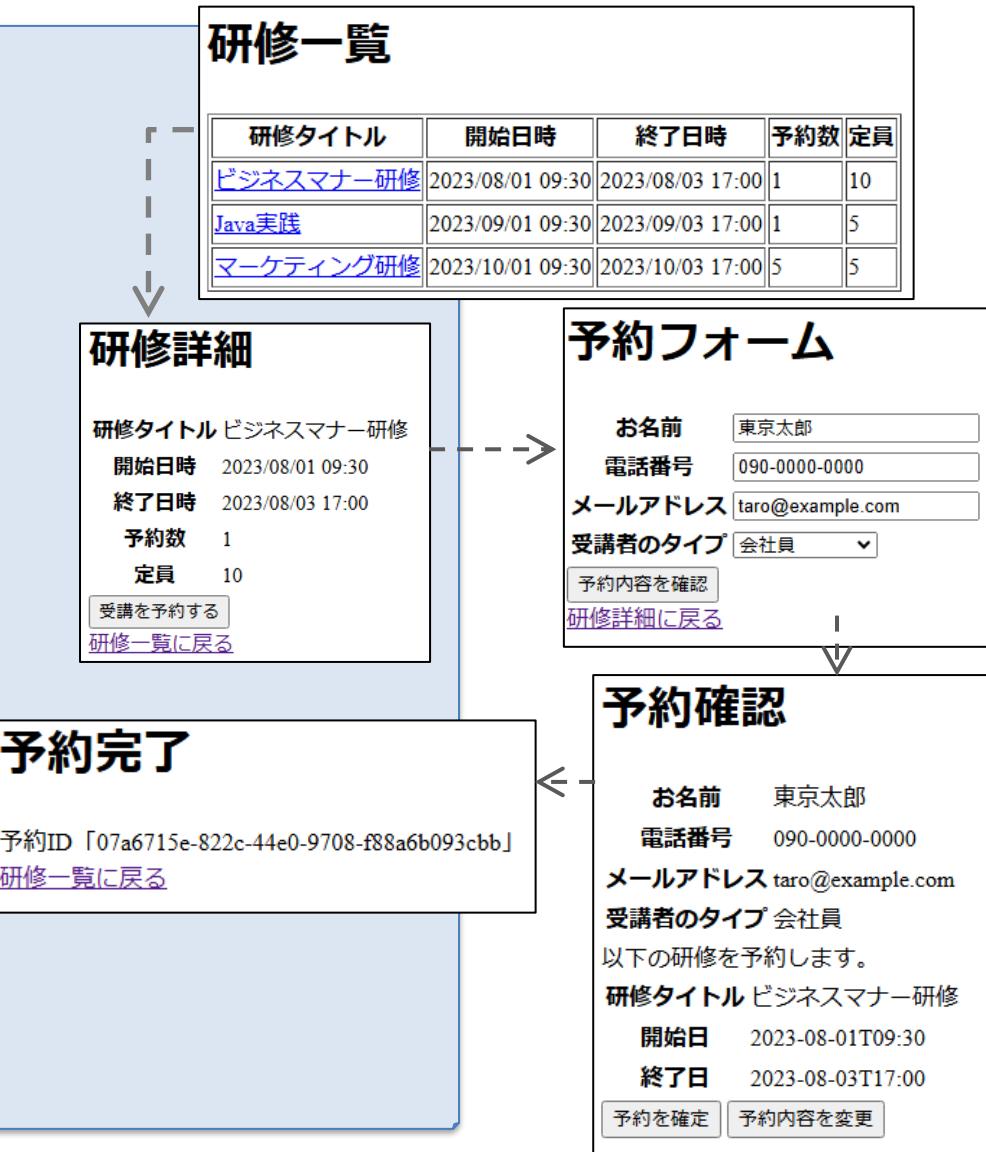
```
<tr>
  <th>研修タイトル</th>
  <td><span>Java研修</span></td>
</tr>
```

```
$( "tr span" ).should( exactText("Java研修") )
```

tr要素の下のspan要素のテキストが「Java研修」になっていることをアサーション

# テストメソッドのサンプル

```
@Test  
@Sql("ReservationUiTest.sql")  
@Sql(value = "clear.sql", executionPhase = ExecutionPhase.AFTER_TEST_METHOD)  
void test_予約する() {  
    open("/training/display-list"); ...①  
    // 研修一覧画面  
    $$("tr span").get(0).should(exactText("ビジネスマナー研修")); ...②  
    ...  
    $$("tr a").get(0).click(); ...③  
    // 研修詳細画面  
    $$("tr span").get(0).should(exactText("ビジネスマナー研修")); ...④  
    $$("tr span").get(1).should(exactText("2023/08/01 09:30")); ...④  
    ...  
    $("input[value=受講を予約する]").click(); ...⑤  
    // 予約フォーム画面  
    $("input[name=name]").setValue("東京太郎"); ...⑥  
    ...  
    $("input[value=予約内容を確認]").click(); ...⑦  
    // 予約確認画面  
    ...  
    $("input[value=予約を確定]").click(); ...⑧  
    // 予約完了画面  
    String reservationId = $($("div span").text()); ...⑨  
    Map<String, Object> reservationMap  
        = jdbcTemplate.queryForMap("SELECT * FROM reservation WHERE id=?", reservationId); ...⑩  
    Assertions.assertThat(reservationMap.get("name")).isEqualTo("東京太郎");  
    ...  
}
```



# テストメソッドのサンプル

①	openメソッドを呼び出してブラウザを起動し、研修一覧画面を表示
②	研修一覧画面の中の0番目のspan要素のテキストが「ビジネスマナー研修」であることをアサーション
③	アンカータグのリンクをクリックし、研修詳細画面を表示
④	研修詳細画面の中の0番目のspan要素のテキストが「ビジネスマナー研修」であることをアサーション
⑤	「受講を予約する」というボタンをクリックして予約フォーム画面を表示
⑥	入力フィールドに「東京太郎」を入力
⑦	「予約内容を確認」というボタンをクリックして予約確認画面を表示
⑧	「予約を確定」ボタンをクリックして完了画面を表示。完了画面の中には、発行された予約IDが表示されている想定になっている
⑨	予約IDの値を画面から取得
⑩	取得した予約IDを使ってデータベースを検索し、期待するレコードが取得されていることを確認

# ハンズオン

---

- ・「2302-shopping-selenide」

# フラッシュスコープの扱い

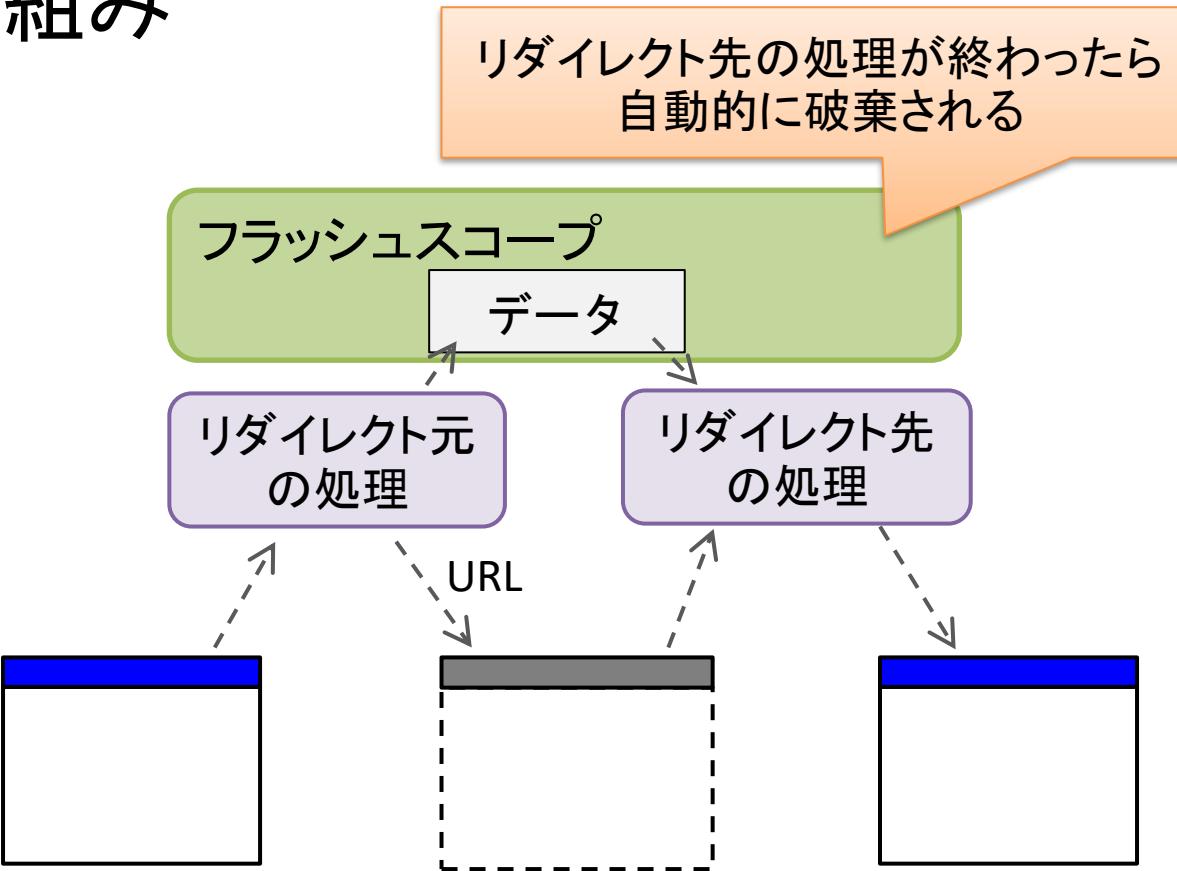
# この章の目標

---

- ・ フラッシュスコープのデータをアサーションできる
- ・ フラッシュスコープのデータを参照できる

# フラッシュスコープとは？

- リダイレクト元とリダイレクト先のリクエストでデータを共有する  
Spring独自の仕組み



# フラッシュスコープを使用するController

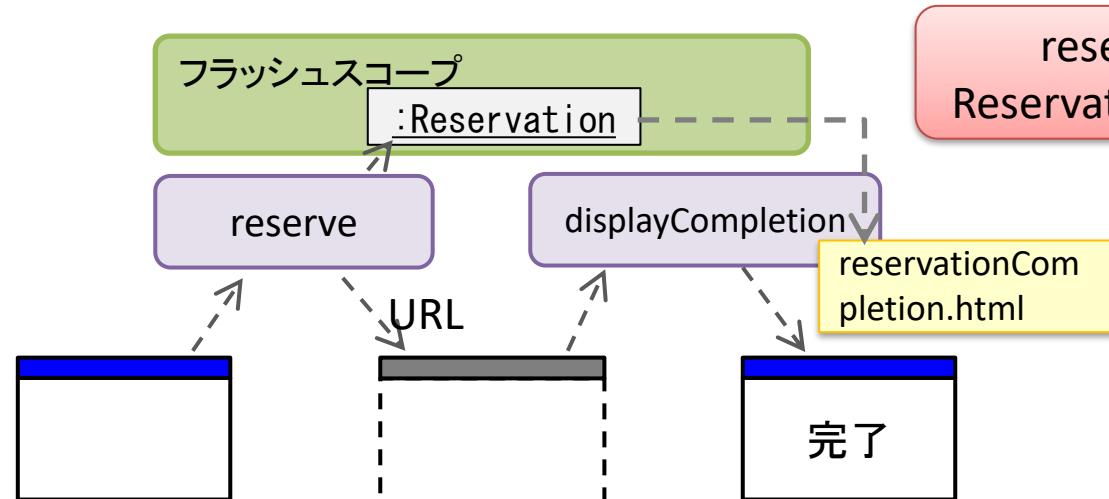
リダイレクト元

```
@PostMapping(value = "/reservation/reserve", params = "reserve")
public String reserve(@Validated ReservationInput reservationInput, RedirectAttributes redirectAttributes) {
    Reservation reservation = reservationService.reserve(reservationInput);
    redirectAttributes.addFlashAttribute("reservation", reservation);
    return "redirect:/reservation/display-completion";
}
```

フラッシュスコープにReservationオブジェクトを格納。

リダイレクト先

```
@GetMapping("/reservation/display-completion")
public String displayCompletion() {
    return "reservation/reservationCompletion";
}
```



reserveメソッドのテストでは、フラッシュスコープに Reservationオブジェクトが格納されていることを確認したい

# フラッシュスコープのデータのアサーション

- MockMvcを使って、フラッシュスコープの中のデータをアサーションできる

```
@WebMvcTest(ReservationController.class)
public class ReservationControllerTest {
    ...
    @MockBean
    ReservationService reservationService;
    @Test
    public void test_reserve() throws Exception {
        Reservation reservation = new Reservation();
        doReturn(reservation).when(reservationService).reserve(any());

        mockMvc.perform(
            post("/reservation/reserve").param("reserve", "")
                .param("name", "東京太郎")
                .param("phone", "090-0000-0000")
                ...
        )
        .andExpect(flash().attribute("reservation", reservation)) ...①
        .andExpect(redirectedUrl("/reservation/display-completion")) ...②
    }
}
```

①	フラッシュスコープの中のデータをアサーション
②	リダイレクトURLの値をアサーション

# フラッシュスコープのデータの参照

```
@SpringBootTest  
...  
public class ReservationControllerIntegrationTest {  
    ...  
    @Autowired  
    JdbcTemplate jdbcTemplate; ...①  
    @Test  
    public void test_reserve() throws Exception {  
        ...  
        MvcResult mvcResult = mockMvc.perform(  
            post("/reservation/reserve").param("reserve", "")  
            .param("name", "東京太郎")  
            .param("phone", "090-0000-0000")  
            ...  
        )  
        ...  
        .andReturn(); ...②  
  
        Reservation reservation = (Reservation) mvcResult.getFlashMap().get("reservation"); ...③  
        Map<String, Object> reservationMap = jdbcTemplate.queryForMap(  
            "SELECT * FROM reservation WHERE id=?", reservation.getId()); ...④  
        assertThat(reservationMap.get("name")).isEqualTo("東京太郎");  
        assertThat(reservationMap.get("phone")).isEqualTo("090-0000-0000");  
    }  
}
```

reserveメソッドのインテグレーションテスト。  
データベースに期待通りに予約データが登録されたかを確認したい

フラッシュスコープの中のReservationオブジェクトから予約IDを取得し、データベースを検索したい

# フラッシュスコープのデータの参照

①	更新されたデータベースの内容を確認するために、JdbcTemplateオブジェクトをインジェクション
②	andReturnメソッドを呼び出して、レスポンスの情報を保持するMvcResultオブジェクトを取得
③	MvcResultオブジェクトが持つgetFlashMapメソッドを呼び出してフラッシュスコープに格納されたデータにアクセスし、Reservationオブジェクトを取得
④	ReservationオブジェクトからIDを取得し、SELECT文のパラメータとして使用。後続の行では、SELECT文で取得したレコードの情報が、期待通りになっていることを確認

# ハンズオン

---

- ・「2402-shopping-test-flash」

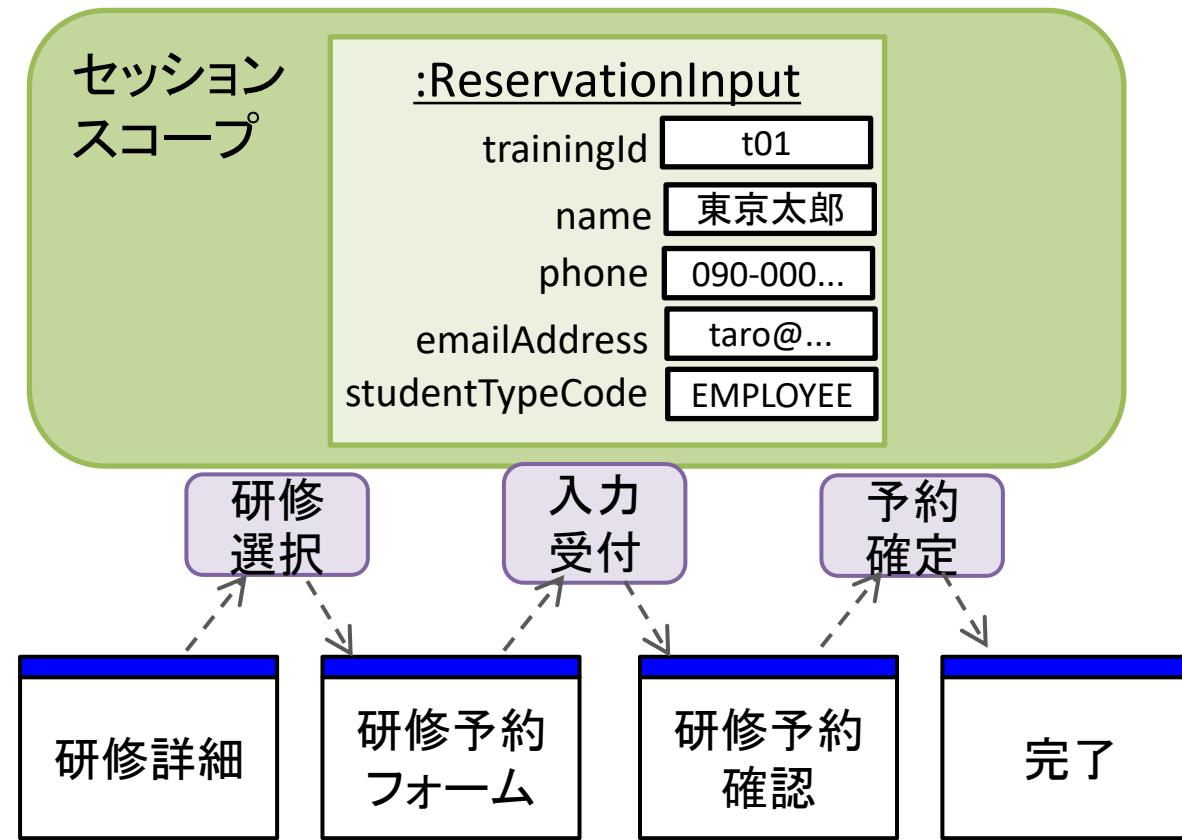
# **セッションスコープのBeanの扱い**

# この章の目標

- ・ セッションスコープのBeanにデータが格納されたことを確認できる
- ・ セッションスコープのBeanが保持するデータを使用していることを確認できる

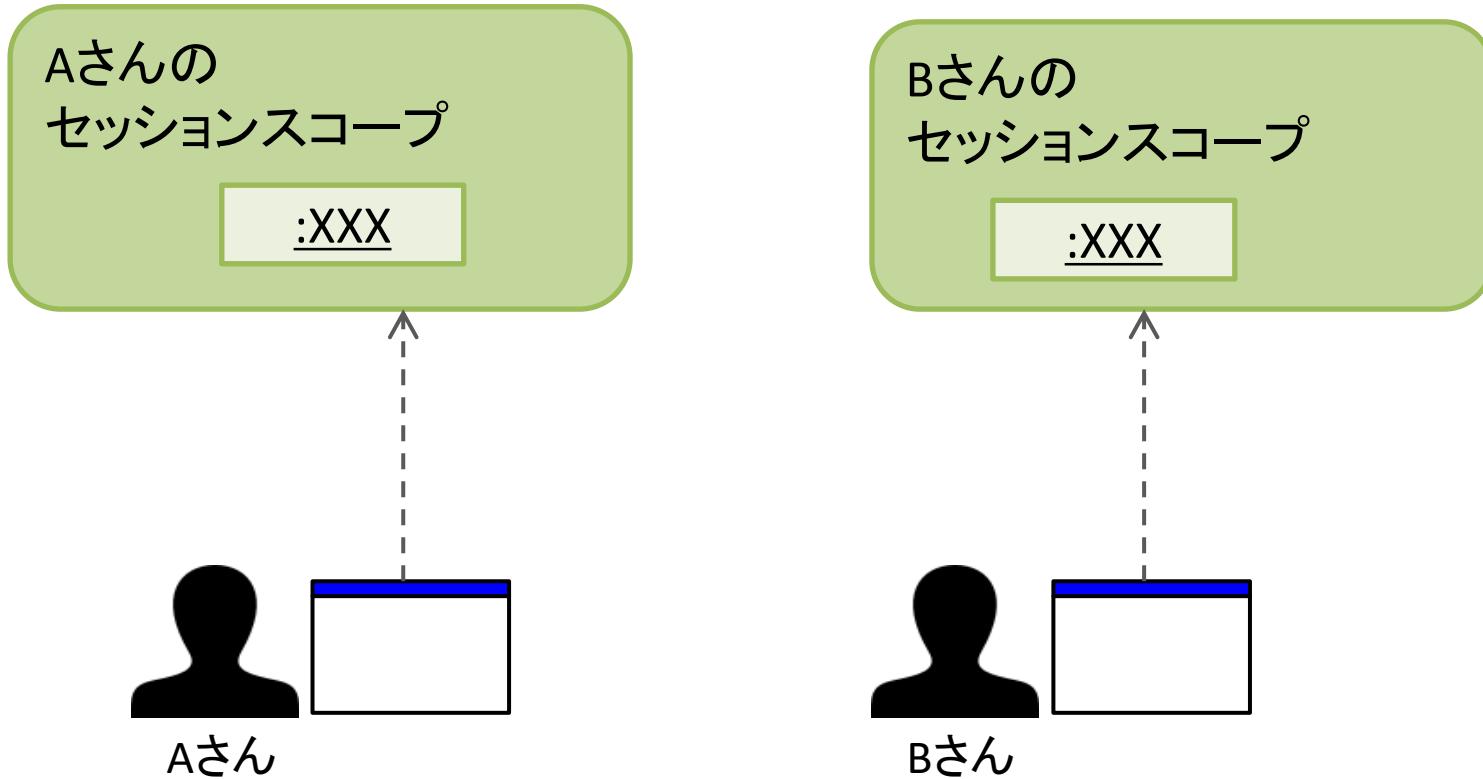
# セッションスコープとは？

- 複数のリクエストに跨ってサーバ側でデータを保持する仕組み
- APサーバが提供する機能



# セッションスコープとは？

- ・アクセスしているブラウザごとにセッションスコープが用意される



# セッションスコープのBean

- セッションスコープごとに個別のオブジェクトを管理してくれるBean

```
@Component("reservationSession") ...①
@SessionScope ...②
@SuppressWarnings("serial")
public class ReservationSession implements Serializable {
    private ReservationInput reservationInput;
    public ReservationInput getReservationInput() {
        return reservationInput;
    }
    public void setReservationInput(ReservationInput reservationInput) {
        this.reservationInput = reservationInput;
    }
    ...
}
```

Aさんの  
セッションスコープ

:ReservationSession  
:ReservationInput

Bさんの  
セッションスコープ

:ReservationSession  
:ReservationInput

- |   |  |
|---|--|
| ① | Bean定義するためのステレオタイプアノテーション。テスト時にBean名を使用するため(後述)、Bean名を明示的にしている。              |
| ② | セッションスコープのBeanを登録するためのアノテーション。セッションスコープの中にReservationSessionオブジェクトが自動的に格納される |

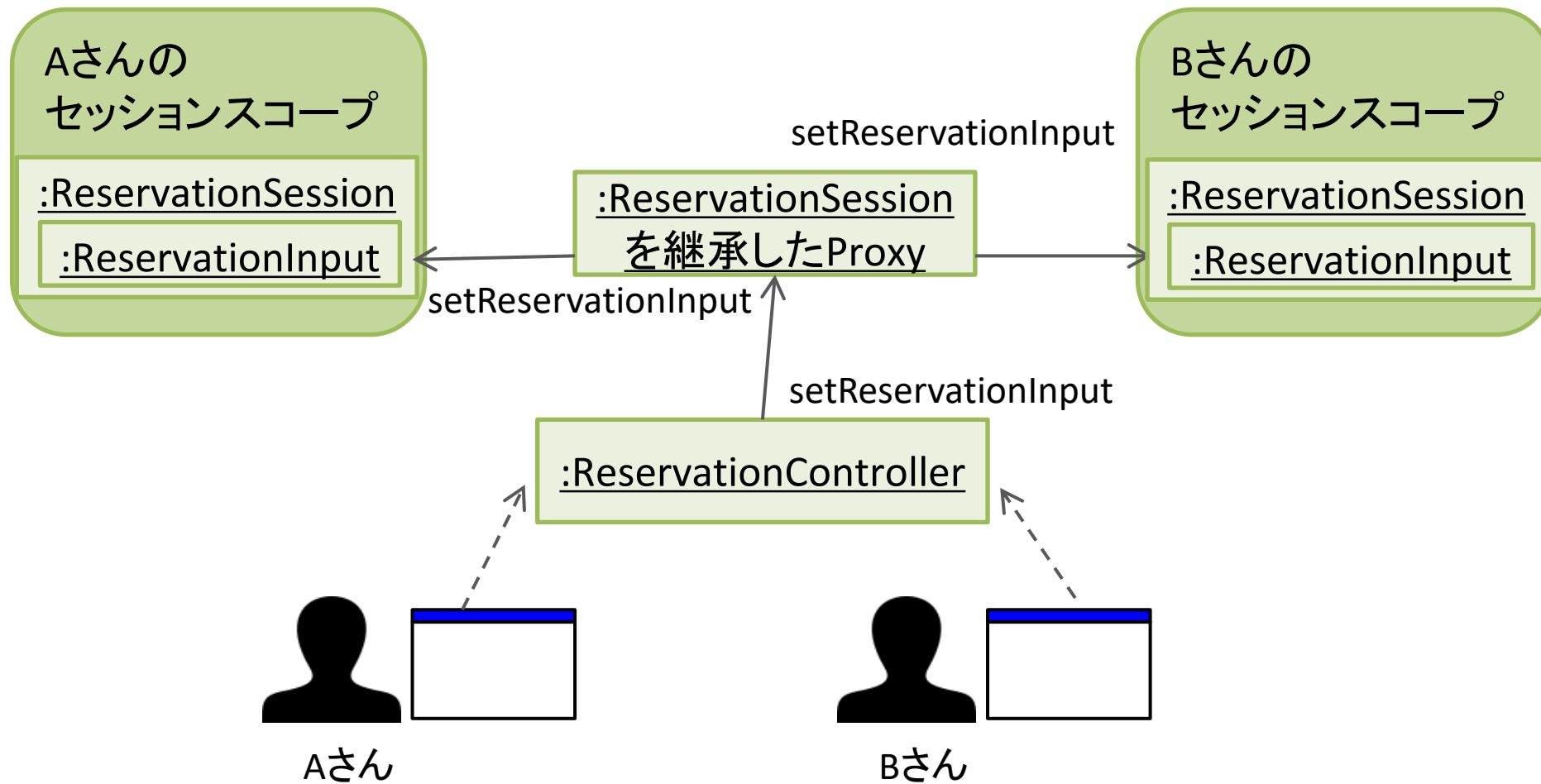
# セッションスコープのBeanを使用するController

```
@Controller  
public class ReservationController {  
    ...  
    private final ReservationSession reservationSession; ...①  
    public ReservationController(ReservationSession reservationSession, ...  
        this.reservationSession = reservationSession;  
    ...  
    @PostMapping("/reservation/validate-input")  
    public String validateInput(@Validated ReservationInput reservationInput, BindingResult bindingResult, Model model) {  
        if (bindingResult.hasErrors()) {  
            ...  
        }  
        ...  
        reservationSession.setReservationInput(reservationInput); ...②  
        return "reservation/reservationConfirmation";  
    }  
    ...
```

セッションスコープごとに個別の  
ReservationSessionオブジェクトになっ  
ている？

①	ReservationSessionオブジェクトをインジェクション
②	ReservationSessionオブジェクトにReservationInputオブジェクトを 格納

# セッションスコープのBeanの仕組み



# セッションスコープのBeanにデータが格納されたことを確認

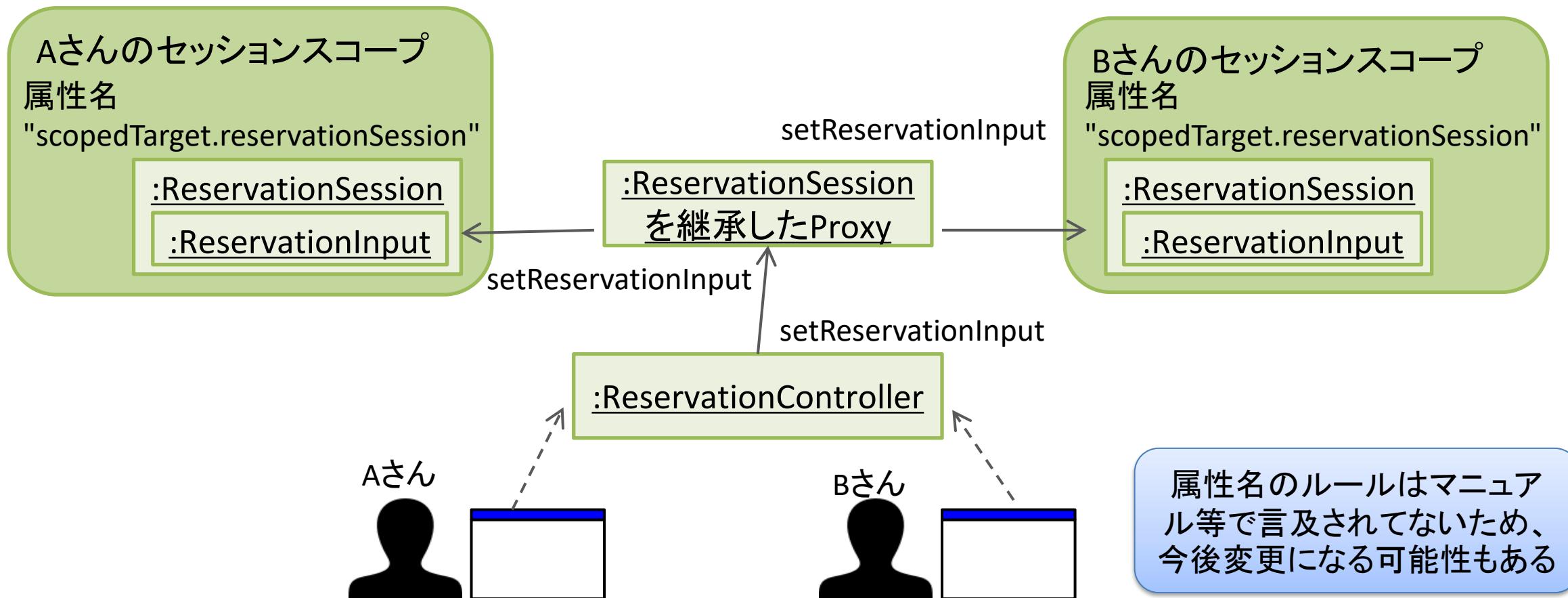
- テスト対象のControllerのメソッド

```
...
@PostMapping("/validate-input")
public String validateInput(@Validated ReservationInput input, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "reservation/reservationForm";
    }
reservationSession.setReservationInput(input);
    return "reservation/reservationConfirmation";
}
...
```

テストメソッドから、セッションスコープ内のReservationSessionオブジェクトを取得し、ReservationInputオブジェクトが期待通りに格納されていることを確認したい

# セッションスコープのBeanの属性名

- セッションスコープのBeanは、属性名「"scopedTarget." + Bean名」で自動的にセッションスコープの中に格納される



# テストのサンプル

```
@WebMvcTest(ReservationController.class)
@Import(ReservationSession.class) ...①
public class ReservationControllerTest {
    ...
    @Test
    void test_validateInput() throws Exception {
        ...
        MvcResult mvcResult = mockMvc.perform(
            post("/reservation/validate-input")
                .param("name", "東京太郎")
                .param("phone", "090-0000-0000")
            ...
        )
        .andExpect(view().name("reservation/reservationConfirmation"))
        .andReturn();
    ;
    ReservationSession reservationSession =
        (ReservationSession)mvcResult.getRequest().getSession().getAttribute("scopedTarget.reservationSession"); ...②
    ReservationInput reservationInput = reservationSession.getReservationInput();
    assertThat(reservationInput.getName()).isEqualTo("東京太郎");
    assertThat(reservationInput.getPhone()).isEqualTo("090-0000-0000");
    ...
}
```

- |   |  |
|---|--|
| ① | コンポーネントスキャンが制限されてReservationSessionクラスに付けた@Componentが無視されるため、明示的にインポート |
| ② | セッションスコープに格納されたReservationSessionオブジェクトを取得                             |

# セッションスコープのBeanが保持するデータを使用していることを確認

- テスト対象のControllerのメソッド

```
...
@PostMapping(value = "/reserve", params = "reserve")
public String reserve(RedirectAttributes redirectAttributes) {
    ReservationInput reservationInput = reservationSession.getReservationInput();
    Reservation reservation = reservationService.reserve(reservationInput);
    redirectAttributes.addFlashAttribute("reservation", reservation);
    return "redirect:/reservation/display-completion";
}
...
```

テストメソッドでセッションスコープに格納しておいた  
ReservationSessionオブジェクトの中のReservationInputオブジェクトが、きちんと使われたことを確認したい

# テストのサンプル

```
@MockBean  
ReservationService reservationService;  
...  
@Test  
public void test_reserve() throws Exception {  
    ...  
    ReservationInput reservationInput = new ReservationInput();  
    reservationInput.setName("東京太郎");  
    ReservationSession reservationSession = new ReservationSession();  ...①  
    reservationSession.setReservationInput(reservationInput);  
    mockMvc.perform(  
        post("/reservation/reserve")  
            .param("reserve", "")  
            .sessionAttr("scopedTarget.reservationSession", reservationSession)  ...②  
    )  
        .andExpect(flash().attribute("reservation", reservation))  
        .andExpect(redirectedUrl("/reservation/display-completion"))  
    ;  
    ArgumentCaptor<ReservationInput> captor = ArgumentCaptor.forClass(ReservationInput.class);  
    verify(reservationService).reserve(captor.capture());  
    ReservationInput capturedInput = captor.getValue();  
    assertThat(capturedInput).isEqualTo(reservationInput);  
}
```

- |   |   |
|---|---|
| ① | セッションスコープに格納するReservationSessionオブジェクトを用意。<br>Setterメソッドを使ってReservationInputオブジェクトを設定 |
| ② | MockMvcでリクエストを送信する際に、ReservationSessionオブジェクト<br>をあらかじめセッションスコープに格納                   |
| ③ | ReservationServiceのreserveメソッドの引数にReservationInputオブジ<br>エクトが渡されたことを確認                |

テキストではControllerのユニットテストのサンプル  
になっているが、Controller・Service・Repositoryのイ  
ンテグレーションテストでも要領は変わらない

# ハンズオン

---

- ・「2502-shopping-test-session」

# 総合演習

# 綜合演習

- 「4002-shopping-comprehensive-exercise」

© 2021-2024 合同会社現場指向

本資料を無断で転載・公開することはご遠慮ください。