Spring Boot 에서 RESTful 에 대해

Spring Boot 에서 RESTful 은 웹 애플리케이션을 구성할 때 REST (Representational State Transfer) 원칙에 따라 클라이언트와 서버 간의 통신을 구현하는 방법을 의미합니다. RESTful API 는 데이터를 쉽게 교환하고 확장 가능한 구조를 지원하는 방식으로 설계되며, 클라이언트-서버 아키텍처에서 주로 JSON 형식의 데이터를 주고받는 데 활용됩니다.

REST 의 주요 원칙

REST 아키텍처는 자원(Resource)을 기반으로 하며, 클라이언트는 자원에 대한 표현(Representation)을 요청하고, 서버는 상태(State)를 전달하는 방식으로 동작합니다. RESTful 의 주요 원칙은 다음과 같습니다:

1. **자원 기반 구조(Resource-based Architecture):**

- 각 자원은 URL(Uniform Resource Locator)을 통해 고유하게 식별됩니다. 예를 들어, `/users`는 사용자 목록을 나타내며, `/users/{id}`는 특정 사용자를 식별합니다.

2. **무상태성(Stateless):**

- RESTful API 는 **무상태**성을 가집니다. 서버는 각 요청을 독립적으로 처리하며, 이전 요청에 대한 정보를 저장하지 않습니다. 따라서 모든 요청은 필요한 모든 정보를 포함해야 하며, 클라이언트는 필요할 때마다 자원에 접근할 수 있습니다.

3. **표준 HTTP 메서드 사용:**

- RESTful API 는 자원에 대한 작업을 HTTP 메서드(GET, POST, PUT, DELETE 등)를 통해 수행합니다.
- **GET**: 자원 조회, **SELECT** 조회
- **POST**: 자원 생성, **INSERT** 처리
- **PUT**: 자원 전체 업데이트, **UPDATE** 처리
- **PATCH**: 자원의 일부 업데이트, **UPDATE**
- **DELETE**: 자원 삭제, **DELETE** 처리

4. **계층적 구조**:

- REST 는 클라이언트와 서버 사이에 프록시, 로드 밸런서 등의 중간 계층이 있어도 문제없이 동작합니다. 클라이언트는 서버와의 직접 통신인지, 중간 계층을 거친 통신인지 구분하지 않아도 됩니다.

5. **캐시 처리(Cacheability):**
- RESTful 서비스는 캐시를 통해 응답 성능을 최적화할 수 있습니다. HTTP 캐시 헤더(`Cache-Control`, `ETag` 등)를 통해 클라이언트가 캐싱된 데이터를 사용할 수 있도록 합니다.

Spring Boot 에서 RESTful API 구현

Spring Boot 는 RESTful API 를 간단하게 구현할 수 있는 다양한 기능을 제공합니다.

1. REST 컨트롤러 작성

Spring Boot 에서는 `@RestController` 어노테이션을 사용하여 RESTful 컨트롤러를 정의합니다.
`@RestController`는 JSON 또는 XML 형식으로 응답을 직렬화하여 반환합니다.

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

@Autowired
private UserService userService;

// GET: 모든 사용자 조회
@GetMapping
public List<User> getAllUsers() {
return userService.getAllUsers();
}

// GET: ID 로 사용자 조회
@GetMapping("/{id}")
public ResponseEntity<User> getUserById(@PathVariable Long id) {
User user = userService.getUserById(id);
return user != null ? ResponseEntity.ok(user) : ResponseEntity.notFound().build();
}
```

```
// POST: 사용자 생성
@PostMapping
public ResponseEntity<User> createUser(@RequestBody User user) {
User createdUser = userService.createUser(user);
return ResponseEntity.status(HttpStatus.CREATED).body(createdUser);
}

// PUT: 사용자 업데이트
@PutMapping("/{id}")
public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User user) {
User updatedUser = userService.updateUser(id, user);
return updatedUser != null ? ResponseEntity.ok(updatedUser) : ResponseEntity.notFound().build();
}

// DELETE: 사용자 삭제
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
userService.deleteUser(id);
return ResponseEntity.noContent().build();
}
}
```

JSP 는 RESTful API 요청에 대해 AJAX 를 사용해야 한다.

1. Fetch All Users (GET /api/users)

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script>
    $(document).ready(function() {
        $.ajax({
            url: '/api/users',
            type: 'GET',
            success: function(users) {
```

```
            console.log("All users:", users);
            // Render users on page
            users.forEach(user => {
                $('#userList').append(`<li>${user.name} - ${user.email}</li>`);
            });
        },
        error: function(xhr, status, error) {
            console.error("Error fetching users:", error);
        }
    });
});
</script>
<ul id="userList"></ul>
```

2. Fetch User by ID (GET /api/users/{id})

```
<script>
    function getUserById(userId) {
        $.ajax({
            url: `/api/users/${userId}`,
            type: 'GET',
            success: function(user) {
                console.log("User:", user);
                // Display user details
                $('#userInfo').text(`Name: ${user.name}, Email: ${user.email}`);
            },
            error: function(xhr, status, error) {
                console.error("Error fetching user by ID:", error);
            }
        });
    }
</script>
```

```
<div id="userInfo"></div>
```

3. Create User (POST /api/users)

```
<script>
    function createUser() {
        const newUser = {
            name: $('#name').val(),
            email: $('#email').val()
        };

        $.ajax({
            url: '/api/users',
            type: 'POST',
            contentType: 'application/json',
            data: JSON.stringify(newUser),
            success: function(user) {
                console.log("User created:", user);
                alert("User created successfully");
            },
            error: function(xhr, status, error) {
                console.error("Error creating user:", error);
            }
        });
    }
</script>
Name: <input type="text" id="name"><br>
Email: <input type="text" id="email"><br>
<button onclick="createUser()">Create User</button>
```

4. Update User (PUT /api/users/{id})

```
<script>
    function updateUser(userId) {
        const updatedUser = {
            name: $('#name').val(),
            email: $('#email').val()
        };

        $.ajax({
            url: `/api/users/${userId}`,
            type: 'PUT',
            contentType: 'application/json',
            data: JSON.stringify(updatedUser),
            success: function(user) {
                console.log("User updated:", user);
                alert("User updated successfully");
            },
            error: function(xhr, status, error) {
                console.error("Error updating user:", error);
            }
        });
    }
</script>
Name: <input type="text" id="name"><br>
Email: <input type="text" id="email"><br>
<button onclick="updateUser(userId)">Update User</button>
```

5. Delete User (DELETE /api/users/{id})

```
<script>
    function deleteUser(userId) {
        $.ajax({
            url: `/api/users/${userId}`,
```

```
            type: 'DELETE',
            success: function() {
                console.log("User deleted");
                alert("User deleted successfully");
            },
            error: function(xhr, status, error) {
                console.error("Error deleting user:", error);
            }
        });
    }
</script>
<button onclick="deleteUser(userId)">Delete User</button>
```

2. HTTP 메서드와 매핑

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`과 같은 어노테이션을 사용하여 HTTP 메서드에 따라 요청을 매핑합니다. 이를 통해 RESTful 방식의 URL 구조와 동작을 명확하게 정의할 수 있습니다.

3. 응답 및 예외 처리

Spring Boot 에서는 `ResponseEntity`를 사용해 응답 코드와 데이터를 유연하게 반환할 수 있습니다. `ResponseEntity`를 활용하여 성공(200 OK), 자원 생성 성공(201 Created), 자원 없음(404 Not Found) 등의 HTTP 상태 코드를 클라이언트에 전달합니다.

예외 처리를 위해 `@ExceptionHandler`를 사용하여 특정 예외가 발생했을 때 적절한 HTTP 상태 코드와 메시지를 반환할 수 있습니다.

```java
```

```
@RestControllerAdvice
public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<String> handleResourceNotFound(ResourceNotFoundException ex) {
return ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
}
}
```

4. 데이터 검증 및 유효성 검사

Spring Boot 에서는 `@Valid`와 `@RequestBody`를 결합하여 데이터를 유효성 검사할 수 있습니다. 유효성 검사가 실패하면 `MethodArgumentNotValidException`이 발생하며, 이를 `@ExceptionHandler`로 처리할 수 있습니다.

```java
@PostMapping
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
User createdUser = userService.createUser(user);
return ResponseEntity.status(HttpStatus.CREATED).body(createdUser);
}
```

```

#### 5. RESTful API 문서화

Swagger 와 같은 도구를 활용하면 API 문서화를 쉽게 할 수 있습니다. Spring Boot 프로젝트에 Swagger 를 통합하여 API 명세서를 자동으로 생성하고 테스트할 수 있습니다.

### RESTful API 의 장점

- **유연성**: JSON, XML 등 다양한 데이터 형식을 지원합니다.
- **확장성**: 자원 기반 구조로 새로운 자원을 쉽게 추가할 수 있습니다.
```

- **보안 및 성능 최적화**: 캐시를 통한 성능 최적화와 계층적 보안을 지원합니다.
- **호환성**: 클라이언트와 서버의 독립성을 보장하여 서로 다른 플랫폼 간 통신이 가능합니다.

RESTful API 의 단점

- **과다한 요청**: 복잡한 작업을 위해 여러 개의 API 호출이 필요할 수 있습니다.
- **데이터 과다 전송**: HTTP 헤더와 상태 코드 등으로 인해 데이터 전송량이 증가할 수 있습니다.
- **추가적인 보안 처리 필요**: 민감한 데이터가 포함된 경우 JWT, OAuth 와 같은 보안 조치가 필요합니다.

Spring Boot 에서 RESTful API 는 웹 애플리케이션과 모바일 애플리케이션에 대한 백엔드 API 를 구축할 때 주로 사용되며, 간결하고 유지보수성이 뛰어난 구조로 클라이언트와 서버 간의 데이터 통신을 효율적으로 지원합니다.