11. RESTful API 설계

REST API 원칙

REST API 원칙(RESTful API Design Principles)은
웹 기반 시스템에서 **일관된 방식으로 자원을 표현하고 조작**할 수 있도록 정의된 설계 기준이다.
REST는 **Representational State Transfer**의 약자로,
클라이언트-서버 간 통신을 **단순하고 예측 가능하게 만들기 위해 고안된 아키텍처 스타일**이다.

✅ 1. 리소스(Resource)는 URI로 식별되어야 한다

원칙	설명
URI는 **명사** 형태로 표현	예: `/users`, `/orders/123`
동사 사용 금지	❌ `/getUser`, `/createOrder` → ❌

> URI는 **"무엇"**을 나타내야 하고, **"무엇을 할지"**는 HTTP Method로 표현해야 함

✅ 2. HTTP 메서드는 의미에 맞게 사용해야 한다

HTTP Method	의미	예시
`GET`	조회 (Read)	`GET /users/1`
`POST`	생성 (Create)	`POST /users`
`PUT`	전체 수정 (Replace)	`PUT /users/1`
`PATCH`	부분 수정 (Partial Update)	`PATCH /users/1`
`DELETE`	삭제	`DELETE /users/1`

> REST의 핵심은 **HTTP 메서드에 의미를 부여하여 행위 표현**

✅ 3. 리소스 간 계층 구조를 표현한다

```
/users/{userId}/orders/{orderId}
```

- 특정 유저의 주문을 표현
- **리소스 간 포함 관계가 명확**하게 드러남

✅ 4. 상태(State)는 클라이언트에 저장 (Stateless)

- 서버는 **요청 간 클라이언트의 상태를 저장하지 않음**
- **모든 요청은 독립적이며, 요청에 필요한 정보를 모두 포함**해야 함

> 인증 토큰(JWT 등)도 요청 헤더에 포함되어야 함

✅ 5. HTTP 상태 코드를 명확하게 사용한다

상태 코드	의미	사용 상황
`200 OK`	성공 (조회, 수정 등)	`GET`, `PUT`, `PATCH`, `DELETE` 성공
`201 Created`	생성 성공	`POST` 성공 시
`204 No Content`	성공 + 응답 없음	삭제 등
`400 Bad Request`	잘못된 요청	유효성 실패, 누락
`401 Unauthorized`	인증 실패	로그인 안 함
`403 Forbidden`	인가 실패	권한 없음
`404 Not Found`	리소스 없음	없는 ID 접근
`500 Internal Server Error`	서버 에러	처리 중 예외 발생

✅ 6. 응답 형식은 JSON + 명확한 구조로

```
{
  "data": {
    "id": 1,
    "name": "Kim",
    "email": "kim@example.com"
  },
  "status": 200,
  "message": "조회 성공"
}
```

> 상태 코드 외에도 **앱 로직에 대한 상태를 명확히 전달**하는 구조를 추천

✅ 7. 에러 응답도 일관된 구조로

```
{
  "error": {
    "status": 400,
    "message": "이메일은 필수 입력 항목입니다.",
    "code": "EMAIL_REQUIRED"
  }
}
```

> 프론트엔드가 **자동 처리 가능하도록 에러 구조를 표준화**

✅ 8. 필터, 정렬, 페이징은 Query Parameter로

```
GET /users?page=2&size=20&sort=name,desc&active=true
```

목적	방식
검색	`?name=kim`
정렬	`?sort=name,desc`
페이징	`?page=1&size=10`
필터링	`?status=ACTIVE`

✅ 9. HATEOAS (확장 선택 원칙)

> Hypermedia As The Engine Of Application State
> 응답에 링크를 포함시켜, 클라이언트가 **다음 행동을 유도**할 수 있도록 설계

```
{
  "data": {...},
  "_links": {
    "self": "/users/1",
    "update": "/users/1",
    "delete": "/users/1"
  }
}
```

> REST 본래 철학이지만, 최근에는 **선택적으로 적용**

✅ 10. RESTful URI 설계 예시

행위	URI	Method
사용자 목록 조회	`/users`	`GET`
사용자 생성	`/users`	`POST`
사용자 조회	`/users/{id}`	`GET`
사용자 수정	`/users/{id}`	`PUT` or `PATCH`
사용자 삭제	`/users/{id}`	`DELETE`
사용자 주문 조회	`/users/{id}/orders`	`GET`

✅ 결론 요약

원칙	설명
리소스는 URI로 명확히 식별	`/users`, `/orders/1`
행위는 HTTP 메서드로 표현	`GET`, `POST`, `PUT`, `DELETE`
URI는 명사형으로	❌ `/getUser` → ✅ `/users`
Stateless 설계	클라이언트가 모든 상태 보관
응답 구조	JSON 표준화 + 상태 코드 사용
검색/정렬/필터	Query Parameter로 표현
HATEOAS	링크를 통해 다음 동작 안내 (선택)

URI 설계 지침

REST API의 **URI 설계**는 단순한 네이밍을 넘어,
서비스의 전체 구조와 리소스 간 관계, 사용성을 결정짓는 핵심 설계 요소이다.
좋은 URI는 **직관적이며, 일관되고, 확장 가능하며, REST 원칙에 충실**해야 한다.

다음은 실무 중심의 **RESTful URI 설계 지침(Best Practices)**이다.

✅ 1. URI는 리소스를 나타내는 명사로 구성

❌ 잘못된 예	✅ 올바른 예
`/getUser/1`	`/users/1`
`/createOrder`	`/orders` (POST)

❌ 잘못된 예	✅ 올바른 예
`/deleteFile`	`/files/{id}` (DELETE)

- URI는 **무엇(What)**을 식별해야 하며,
- **행위(How)**는 **HTTP 메서드**로 구분한다.

✅ 2. 리소스는 복수형 명사로

리소스	URI
사용자	`/users`
상품	`/products`
주문	`/orders`

- REST에서는 `/user`보다는 `/users`가 표준에 가깝다.
- 하위 자원(`users/1`)도 자연스럽게 확장 가능

✅ 3. 계층 구조 표현

```
/users/{userId}/orders/{orderId}
```

- 계층 구조는 **포함 관계**를 표현함
- 예: 특정 사용자에 속한 주문

✅ 4. 파일 형식, 응답 형태는 URI에 포함하지 않는다

```
GET /users/1.json    ❌
GET /users/1         ✅ → Accept: application/json 헤더 사용
```

> MIME 타입은 **Content Negotiation**으로 처리 (`Accept`, `Content-Type`)

✅ 5. 쿼리 파라미터는 검색, 정렬, 필터링에 사용

```
GET /products?category=shoes&sort=price,desc&page=2&size=20
```

목적	방식
검색	`?name=kim`
정렬	`?sort=price,desc`

목적	방식
페이징	`?page=1&size=10`
필터링	`?status=ACTIVE&region=seoul`

✅ 6. 액션은 URI가 아닌 HTTP 메서드로 구분

행위	URI	메서드
사용자 생성	`/users`	`POST`
사용자 조회	`/users/{id}`	`GET`
사용자 수정	`/users/{id}`	`PUT` or `PATCH`
사용자 삭제	`/users/{id}`	`DELETE`

✅ 7. 관계성 있는 리소스는 하위로 연결

```
GET /users/1/orders       # 사용자 1의 주문 목록
GET /users/1/orders/15    # 사용자 1의 주문 상세
```

> 리소스 간 관계가 명확해지고 **URI만으로 의미 파악 가능**

✅ 8. 컨트롤러별 URI는 도메인 기반으로 그룹화

```
GET    /users
POST   /users
GET    /users/{id}
PATCH  /users/{id}

GET    /orders
POST   /orders
```

> 도메인(Resource) 단위로 URI를 구성하고
> **HTTP 메서드**만으로 요청 행위를 구분

✅ 9. URI에는 동사를 포함하지 않는다

❌ 잘못된 예	✅ 올바른 예
`/createUser`	`/users` (POST)
`/deleteUser/1`	`/users/1` (DELETE)

❌ 잘못된 예	✅ 올바른 예
`/updateUser`	`/users/1` (PUT/PATCH)

> URI는 동작이 아닌 **리소스의 경로**를 표현하는 것이 목적

✅ 10. 상태 변경은 HTTP Method로 표현하되, 필요 시 별도 액션 URI 허용

일반적인 CRUD 외 **행위성 작업(승인, 취소 등)**은 다음처럼 표현:

```
POST /orders/123/cancel
POST /accounts/999/activate
```

- REST 철학에선 메서드만으로도 가능하지만,
- 실무에서는 **의미 있는 명시적 경로**로 유지하는 것도 좋음

✅ 11. ID는 URI에 포함, 상세 데이터는 Body로

```
POST /users
{
  "name": "Kim",
  "email": "kim@a.com"
}

PUT /users/1
{
  "name": "Kim Updated"
}
```

✅ 12. 슬래시(`/`) 사용 원칙

규칙	예시
루트 구분자	`/users/1/orders/5`
끝에 `/` 붙이지 않음	`/users/1/` → ❌
계층 표현 시에만 `/` 사용	OK

✅ 13. URI는 소문자, 특수문자는 `-` 사용

```
/users → ✅
/Users → ❌
/user-profile → ✅
/user_profile → ❌ (언더스코어 사용 지양)
```

✅ 14. RESTful URI 설계 패턴 예시

리소스	URI	설명
전체 조회	`GET /users`	사용자 목록
단건 조회	`GET /users/{id}`	사용자 상세
생성	`POST /users`	사용자 생성
전체 수정	`PUT /users/{id}`	전체 필드 교체
일부 수정	`PATCH /users/{id}`	일부 필드 변경
삭제	`DELETE /users/{id}`	사용자 삭제
하위 자원	`GET /users/{id}/orders`	사용자 주문 목록
사용자 상태 변경	`POST /users/{id}/deactivate`	비표준 액션

✅ 결론 요약

원칙	설명
명사 기반	`/users`, `/products` 형태
복수형	리소스는 복수형 사용 권장
동사 금지	`GET /users` vs ❌ `/getUsers`
관계 표현	`/users/1/orders`
검색/정렬	Query 파라미터로
액션은 Method	`POST`, `PUT`, `PATCH`, `DELETE`로 동작 구분
소문자 + `-`	URI 표기 규칙 준수

HATEOAS

HATEOAS(Hypermedia As The Engine Of Application State)는
REST 아키텍처의 핵심 원칙 중 하나로,
클라이언트가 API의 응답 안에 포함된 링크(Hypermedia)를 통해 다음 행동을 스스로 발견하도록 유도하는 개념이다.

즉, 클라이언트가 서버로부터 받는 응답에 **자원뿐만 아니라 해당 자원과 관련된 다른 행위(링크)**도 함께 전달되며,
이를 통해 **클라이언트는 하드코딩 없이도 API를 탐색할 수 있게 된다.**

✅ 1. HATEOAS의 목적

목적	설명
자기 설명(self-descriptive)	응답 자체에 가능한 동작을 설명
유연한 API 진화	클라이언트가 링크만 따라가므로 서버 URI 변경 가능
API 탐색 가능성	클라이언트가 상태 전이 가능한 경로를 알 수 있음
명시적 제어	어떤 사용자가 어떤 행동을 할 수 있는지 서버가 제시

✅ 2. 일반 REST vs HATEOAS 응답

❌ 일반 REST 응답

```
{
  "id": 1,
  "name": "kim",
  "email": "kim@example.com"
}
```

✅ HATEOAS 응답

```
{
  "id": 1,
  "name": "kim",
  "email": "kim@example.com",
  "_links": {
    "self": { "href": "/users/1" },
    "update": { "href": "/users/1", "method": "PUT" },
    "delete": { "href": "/users/1", "method": "DELETE" },
    "orders": { "href": "/users/1/orders" }
  }
}
```

> 클라이언트는 어떤 링크를 클릭하면 어떤 행동을 할 수 있는지 **서버가 안내**한다.

✅ 3. Spring HATEOAS 사용법

📌 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-hateoas'
```

📌 엔티티에 링크 추가

```
@Data
@AllArgsConstructor
public class User {
    private Long id;
    private String name;
}
```

📌 Controller에서 링크 구성

```
@GetMapping("/users/{id}")
public EntityModel<User> getUser(@PathVariable Long id) {
    User user = new User(id, "kim");

    return EntityModel.of(user,
        linkTo(methodOn(UserController.class).getUser(id)).withSelfRel(),
        linkTo(methodOn(UserController.class).deleteUser(id)).withRel("delete"),
        linkTo(methodOn(OrderController.class).getOrdersByUser(id)).withRel("orders")
    );
}
```

📌 반환 결과

```
{
  "id": 1,
  "name": "kim",
  "_links": {
    "self": { "href": "http://localhost:8080/users/1" },
    "delete": { "href": "http://localhost:8080/users/1" },
    "orders": { "href": "http://localhost:8080/users/1/orders" }
  }
}
```

✅ 4. 링크 구성 도우미 설명

메서드	설명
`EntityModel.of()`	자원 + 링크를 하나로 감싸는 래퍼
`linkTo(...)`	해당 컨트롤러 메서드 URI 추출
`methodOn(...)`	메서드 시그니처에 따라 링크 유추
`withRel("delete")`	관계 이름(Relation) 설정
`withSelfRel()`	`self` 링크로 자동 명명

✅ 5. 컬렉션 응답 (리스트)

```
@GetMapping("/users")
public CollectionModel<EntityModel<User>> listUsers() {
    List<User> users = List.of(new User(1L, "kim"), new User(2L, "lee"));

    List<EntityModel<User>> result = users.stream()
        .map(user -> EntityModel.of(user,

linkTo(methodOn(UserController.class).getUser(user.getId())).withSelfRel()))
        .toList();

    return CollectionModel.of(result,
        linkTo(methodOn(UserController.class).listUsers()).withSelfRel());
}
```

✅ 6. HATEOAS 적용 시 이점

항목	설명
명시적 흐름 제시	가능한 다음 행동을 명확히 알려줌
클라이언트 분기 줄임	URI 직접 조합 없이 서버 안내 따름
API 변경 내성	URI 변경에도 클라이언트는 링크 기반으로 탐색
보안/권한 통제	허용된 링크만 포함시키면 됨

✅ 7. 단점 및 현실적 조언

항목	설명
단점	복잡도 증가, 응답 길이 증가, 프론트에서 잘 사용하지 않음
프론트엔드 대응	HATEOAS 기반으로 작성되지 않는 경우가 많음 (SPA는 REST보다 RPC 선호)
적용 전략	**API 자체 문서화로 활용**, 내부 연동 또는 보안 강조 API에 활용 추천

✅ 결론 요약

항목	설명
정의	응답에 링크를 포함해 **다음 행동을 제시하는 REST 확장**
구현 방식	`EntityModel`, `CollectionModel`, `linkTo()`, `methodOn()` 활용
장점	API 흐름 안내, 내구성 향상, 권한 제어

항목	설명
단점	클라이언트 호환성 낮음, 사용 복잡도
실무 팁	Spring REST Docs와 함께 사용 시 문서 자동화 효과 탁월

Swagger / Springdoc OpenAPI 연동

Swagger (OpenAPI)는 REST API의 명세를 문서화하고 테스트할 수 있는 표준 도구이며,
Spring Boot에서는 `springdoc-openapi` 라이브러리를 통해 Swagger UI를 손쉽게 연동할 수 있다.

이 연동을 통해 개발자는 다음과 같은 이점을 얻는다:

- API의 동작을 **문서화하고 시각적으로 확인**
- **테스트와 디버깅**을 UI에서 바로 실행 가능
- **자동화된 스펙 관리**로 팀 협업 효율 증가

✅ 1. 의존성 추가 (Gradle)

```
implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.2.0'
```

> Spring Boot 3 이상에서는 2.x 버전을 사용해야 함 (`springdoc-openapi-starter-*` 패키지)

✅ 2. 기본 연동만으로 가능한 것들

Spring Boot 앱에 위 의존성만 추가하면 자동으로 다음 URL이 활성화된다:

목적	주소
Swagger UI	`http://localhost:8080/swagger-ui.html`
OpenAPI JSON 문서	`http://localhost:8080/v3/api-docs`

→ 컨트롤러에 정의된 모든 API가 **자동으로 문서화**

✅ 3. 기본 컨트롤러 예시

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

    @GetMapping("/{id}")
    public ResponseEntity<UserDto> getUser(@PathVariable Long id) {
        return ResponseEntity.ok(new UserDto(id, "kim", "kim@example.com"));
    }

    @PostMapping
```

```
    public ResponseEntity<UserDto> createUser(@RequestBody UserDto dto) {
        return ResponseEntity.status(HttpStatus.CREATED).body(dto);
    }
}
```

→ 위 API들은 Swagger UI에서 자동으로 확인 가능

✅ 4. @Operation, @Parameter 등 문서 커스터마이징

```
@Operation(summary = "회원 조회", description = "ID로 사용자 정보를 조회합니다.")
@ApiResponses({
    @ApiResponse(responseCode = "200", description = "조회 성공"),
    @ApiResponse(responseCode = "404", description = "사용자 없음")
})
@GetMapping("/{id}")
public ResponseEntity<UserDto> getUser(
    @Parameter(description = "회원 ID", example = "1")
    @PathVariable Long id
) {
    ...
}
```

> 필요한 항목만 선택적으로 추가해도 Swagger 문서가 훨씬 읽기 쉬워짐

✅ 5. @Schema로 DTO 문서화

```
@Data
@Schema(description = "사용자 DTO")
public class UserDto {

    @Schema(description = "사용자 ID", example = "1")
    private Long id;

    @Schema(description = "이름", example = "kim")
    private String name;

    @Schema(description = "이메일 주소", example = "kim@example.com")
    private String email;
}
```

> `@Schema`, `@ArraySchema`, `@Schema(hidden = true)` 등으로 JSON 구조를 명확히 표시 가능

✅ 6. 전체 API 정보 설정 (`OpenAPI` Bean)

```java
@Bean
public OpenAPI customOpenAPI() {
    return new OpenAPI()
        .info(new Info()
            .title("My API 문서")
            .version("v1")
            .description("SpringBoot + Swagger 연동 예시")
        );
}
```

→ Swagger UI 상단에 제목, 버전, 설명 표시됨

✅ 7. 인증 헤더 적용 (JWT 등)

```java
@Bean
public OpenAPI securityOpenAPI() {
    return new OpenAPI()
        .addSecurityItem(new SecurityRequirement().addList("Authorization"))
        .components(new Components().addSecuritySchemes("Authorization",
            new SecurityScheme()
                .name("Authorization")
                .type(SecurityScheme.Type.HTTP)
                .scheme("bearer")
                .bearerFormat("JWT")
                .in(SecurityScheme.In.HEADER)
        ));
}
```

→ Swagger UI에서 토큰을 입력할 수 있는 창이 생성됨
→ 실제 요청에도 `Authorization: Bearer <token>` 헤더 자동 포함

✅ 8. Springdoc 주요 설정 옵션 (application.yml)

```yaml
springdoc:
  default-produces-media-type: application/json
  api-docs:
    enabled: true
    path: /v3/api-docs
  swagger-ui:
    enabled: true
    path: /swagger-ui.html
    operations-sorter: alpha
    tags-sorter: alpha
```

✅ 9. 경로 필터링 (특정 패키지만 문서화)

```
springdoc:
  packages-to-scan: com.example.api.controller
```

> 큰 프로젝트일수록 불필요한 엔드포인트 제외 가능

✅ 10. Spring Security와 함께 사용하는 경우

Security가 `/v3/api-docs`, `/swagger-ui.html` 경로를 차단하면 허용 필수:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/swagger-ui.html", "/swagger-ui/**", "/v3/api-
docs/**").permitAll()
        .anyRequest().authenticated();
}
```

✅ 결론 요약

항목	설명
Swagger UI	`springdoc-openapi`로 자동 연동 (`/swagger-ui.html`)
문서 커스터마이징	`@Operation`, `@Parameter`, `@Schema` 사용
보안 설정	JWT 인증 헤더 UI에 포함 가능
실무 전략	API 테스트 + 개발 협업 + 문서 자동화 3박자 모두 만족
권장 도구	`Swagger UI` + `OpenAPI JSON` → CI/CD에 자동 포함 가능

Postman 테스트 자동화

Postman 테스트 자동화는 REST API 개발 이후 반드시 필요한 **신뢰성과 반복성 있는 테스트를 구축**하는 방법이다.
단순한 수동 요청 전송을 넘어, **API 시나리오 테스트, 자동화된 검증, CI/CD 연동**까지 가능하다.

✅ 1. Postman의 자동화 기능 종류

기능	설명
Tests 탭	요청 응답에 대해 JavaScript로 검증 로직 작성 가능
Collection Runner	여러 API를 시나리오 기반으로 순차 실행
Environment	요청 URL, 토큰 등 변수화 가능 (dev/test/prod)

기능	설명
Newman	Postman Collection을 **CLI에서 실행**하는 도구 (CI 연동 핵심)

✅ 2. 기본 구조 예시: Tests 탭

📌 요청 후 응답 검증 코드 (JavaScript)

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("응답에 name 필드가 존재", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData.name).to.eql("kim");
});
```

📌 조건부 테스트, 값 추출

```
// 토큰 추출 후 변수 저장
let data = pm.response.json();
pm.environment.set("access_token", data.token);
```

→ 다음 요청에서 `{{access_token}}` 변수로 사용 가능

✅ 3. Collection Runner 사용 (GUI)

1. 좌측에서 테스트할 Collection 선택
2. [▶️ Run Collection] 버튼 클릭
3. Environment, 반복 횟수, CSV/JSON 데이터 파일 지정 가능
4. 결과 리포트 제공

> **시나리오 테스트**, **CSV 기반 파라미터화 테스트**에 유용

✅ 4. Postman 환경 구성 예시

변수 이름	값 (dev 예시)
`base_url`	`http://localhost:8080/api`
`access_token`	동적으로 저장

→ 요청에서 `{{base_url}}/users` 형식으로 사용

✅ 5. 자동화 핵심: Newman (CLI 실행기)

📌 설치

```
npm install -g newman
```

📌 실행 예

```
newman run UserAPI.postman_collection.json \
  --environment dev.postman_environment.json \
  --reporters cli,html \
  --reporter-html-export ./newman-report.html
```

옵션	설명
`--environment`	Postman 환경 파일 지정
`--reporters`	결과 리포터 지정 (cli, json, html 등)
`--reporter-html-export`	리포트 파일 저장 경로

📌 CSV 기반 반복 테스트

```
newman run collection.json --iteration-data data.csv
```

> 여러 테스트 데이터를 반복적으로 실행할 수 있음

✅ 6. CI/CD 연동 전략

환경	연동 방식
GitHub Actions	`newman-action`, `npm install && newman run`
GitLab CI	`.gitlab-ci.yml`에 Newman 설치 및 실행 포함
Jenkins	`sh 'newman run ...'` 빌드 단계에 추가
Docker	`postman/newman` 이미지 사용 가능

✅ 7. 리포트 종류

리포터	설명
`cli`	터미널 출력 (기본)

리포터	설명
`json`	기계 분석용 리포트
`html`	사람이 보기 좋은 리포트 (대시보드처럼 사용 가능)
`junit`	JUnit XML → Jenkins 연동용

✅ 8. 실무 활용 팁

항목	설명
테스트 버전 관리	`.postman_collection.json`, `.postman_environment.json` Git에 포함
실패 조건 엄격화	응답 값, 상태 코드, 에러 메시지 등 세밀하게 비교
보안	민감 변수는 Postman Secret 혹은 CI 암호 변수로 분리
분리 설계	각 기능별 Collection 분리 + 공통 환경 재사용
Export 기준	항상 최신 Collection → JSON 파일 export 후 저장

✅ 9. 샘플 테스트 코드 정리

```
pm.test("토큰이 존재해야 한다", function () {
    var json = pm.response.json();
    pm.expect(json.token).to.be.a('string');
});

pm.test("응답 속도는 1초 이내", function () {
    pm.expect(pm.response.responseTime).to.be.below(1000);
});
```

✅ 결론 요약

항목	설명
테스트 작성 위치	Postman 요청의 Tests 탭 (JavaScript 기반)
자동화 실행	Collection Runner (GUI) / Newman (CLI)
변수화	Environment 사용, `{{var_name}}` 형식
CI 연동	GitHub Actions, GitLab, Jenkins 등에서 `newman run`
리포트	HTML, JUnit, JSON 등 다양한 형식 지원
실무 효과	반복 테스트, 품질 보장, 배포 전 검증 자동화