스칼라 주식 시장

작성일: 2025-09-08

작성자: 윤소현

실습 개요

- 목적
  - Spring Boot REST API + Vue.js 프론트엔드 + Docker/K8s 배포를 통한 주식 거래 시스템 생성
- 구현

  1. 백엔드 (Spring Boot)
     - 위치
       - `/Users/yshmbid/Documents/home/github/Spring/basic`
     - 패키지 구조

       `controller`(API), `entity`(DB 테이블 매핑), `repository`(JPA), `service`(비즈니스 로직), `dto`(전송 객체)로 구성
     - 주요 API
       - `/api/stocks` : 주식 CRUD (목록 조회, 상세 조회, 등록, 수정, 삭제)
       - `/api/players` : 플레이어 CRUD, 로그인, 매수/매도, 보유 주식 조회, 잔액·자산 평가, 거래내역 조회 등
     - 추가 기능
       - `@Valid`와 `GlobalExceptionHandler`로 입력값 검증 및 전역 예외 처리
       - H2 DB → 파일 모드로 전환해 데이터 영속화
       - Swagger(OpenAPI)로 API 문서 자동화
       - PlayerBalance, PlayerPortfolio, Transactions 등 확장 API 구현
  2. 프론트엔드 (Vue.js + Vite + Nginx)
     - 위치
       - `/Users/yshmbid/Documents/home/github/frontend`
     - 주요 페이지
       - `HelloView.vue` : 로그인
       - `PlayersView.vue` : 플레이어 관리 (등록, 조회, 매수·매도)
       - `StocksView.vue` : 주식 관리 (등록, 조회, 수정, 삭제)
     - API 연동
       - `src/api/` 폴더 내 `stocks.js`, `players.js` 등으로 백엔드 REST API 호출
  3. 배포 (Docker + Kubernetes)
     - Dockerfile로 백엔드/프론트엔드 각각 이미지 생성
     - Step
       - `docker-build.sh`, `docker-push.sh`로 이미지 빌드 & 레지스트리에 푸시
       - `k8s/deploy.yaml`, `k8s/service.yaml`로 쿠버네티스 배포
       - `kubectl port-forward` 또는 `Ingress`를 통해 외부 접속 확인
       - `https://ingress.skala25a.project.skala-ai.com/sk019-api/redoc.html` API 확인

실습 내용

▼ 1. skala-stock-api 프로젝트 생성

https://github.com/Ismin625/skala-stock-api.git → practice 다운로드 후 압축해제

▼ 2. 패키지 구조

```
skala-stock-api
 ├─ src/main/java/com/sk/skala/stockapi
 │   ├─ com.sk.skala.stockapi.aop
 │   ├─ com.sk.skala.stockapi.config
 │   ├─ com.sk.skala.stockapi.controller
 │   │   ├─ StockController.java
 │   │   └─ PlayerController.java
 │   ├─ com.sk.skala.stockapi.data
 │   │   ├─ Stock.java
 │   │   ├─ Player.java
 │   │   └─ PlayerStock.java
 │   ├─ com.sk.skala.stockapi.dto
 │   │   ├─ PlayerStockDto.java
 │   │   └─ PlayerStockListDto.java
 │   ├─ com.sk.skala.stockapi.exception
 │   ├─ com.sk.skala.stockapi.repository
 │   │   ├─ StockRepository.java
 │   │   ├─ PlayerRepository.java
 │   │   └─ PlayerStockRepository.java
 │   ├─ com.sk.skala.stockapi.service
 │   │   ├─ StockService.java
 │   │   └─ PlayerService.java
 │   └─ com.sk.skala.stockapi.tools
 └─ src/main/resources
     └─ application.yml
```

- com.sk.skala.stockapi - 애플리케이션 기본 패키지
  - com.sk.skala.stockapi.aop
    - AOP 클래스 – API 로그
  - com.sk.skala.stockapi.config
    - 초기 설정 – DB 접속, 보안
  - com.sk.skala.stockapi.controller
    - API 컨트롤러 – API 엔드포인
  - com.sk.skala.stockapi.data
    - 데이터 객체 – 테이블, DTO
  - com.sk.skala.stockapi.exception
    - 예외 클래스
  - com.sk.skala.stockapi.repository
    - DB 레포지토리
  - com.sk.skala.stockapi.service
    - 서비스 – 컨트롤러와 레포지토리 연결
  - com.sk.skala.stockapi.tools
    - 유틸리티 (문자열, JSON, JWT 등)

- src/main/resources
  - application.yml
    - 애플리케이션 구성 정보

▼ 3. API 목록

```
API
 ├─ /api/stocks
 │   ├─ GET /api/stocks
 │   ├─ GET /api/stocks/{id}
 │   ├─ POST /api/stocks
 │   ├─ PUT /api/stocks
 │   └─ DELETE /api/stocks
 │
 └─ /api/players
    ├─ GET /api/players
    ├─ GET /api/players/{playerId}
    ├─ GET /api/players/{playerName}
    ├─ POST /api/players
    ├─ POST /api/players/login
    ├─ PUT /api/players
    ├─ DELETE /api/players/{playerId}
    ├─ GET /api/players/{playerId}/stocks
    ├─ POST /api/players/buy
    └─ POST /api/players/sell
```

- /api/stocks
  - GET /api/stocks
    - 주식 전체 목록 조회
  - GET /api/stocks/{id}
    - 개별 주식 상세 조회
  - POST /api/stocks
    - 주식 정보 추가
  - PUT /api/stocks
    - 주식 정보 변경
  - DELETE /api/stocks
    - 주식 삭제
- /api/players
  - GET /api/players
    - 플레이어 전체 목록 조회
  - GET /api/players/{playerId}, GET /api/players/{playerName}
    - 개별 플레이어 조회, 이름으로 조회
  - POST /api/players
    - 플레이어 등록
  - POST /api/players/login
    - 로그인
  - PUT /api/players

        - 플레이어 정보 변경
    - DELETE /api/players/{playerId}
        - 플레이어 삭제
    - GET /api/players/{playerId}/stocks
        - 플레이어 보유 주식 조회
    - POST /api/players/buy, POST /api/players/sell
        - 주식 매수, 주식 매도

▼ 4. Maven pom.xml

- 프로젝트 정보

```
<groupId>com.sk.skala</groupId>
<artifactId>skala-stock-api</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>skala-stock-api</name>
<description>SKALA STOCK REST API</description>
<properties>
  <java.version>17</java.version>
</properties>
```

Maven이라는 빌드 도구가 프로젝트를 관리하는데, 관리에 필요한 이 프로젝트 정보.

- groupId - com.sk.skala
    - 프로젝트가 속한 그룹 고유 식별자
- artifactId - skala-stock-api
    - skala.sk.com 안에서 프로젝트를 구분하는 이름
- version - 0.0.1-SNAPSHOT
    - 현재 배포 버전.
    - SNAPSHOT은 아직 개발 중이라는 표시고 정식 릴리즈 때는 1.0.0처럼 SNAPSHOT이 빠진다.
- name - skala-stock-api
    - 프로젝트의 이름
    - 실질적인 빌드/의존성 식별에는 groupId, artifactId, version을 쓰긴한데 얘는 그냥 사람이 볼 때 구분하기 쉽도록 하는 용도.
- description - SKALA STOCK REST API
    - 프로젝트 설명
- java.version - 17
    - 프로젝트의 자바 버전 (21이어야대는거아닌가?)

▼ 5. 프로젝트 구조

```
Spring/basic
 └─ src/main/java/com/skala/basic
    ├─ controller
    │   ├─ CourseController.java
    │   ├─ HelloController.java
    │   ├─ UserController.java
    │   ├─ StockController.java
```

```
│   └── PlayerController.java
├── entity
│   ├── Course.java
│   ├── Stock.java
│   ├── Player.java
│   └── PlayerStock.java
├── dto
│   ├── PlayerStockDto.java
│   └── PlayerStockListDto.java
├── repository
│   ├── StockRepository.java
│   ├── PlayerRepository.java
│   └── PlayerStockRepository.java
└── service
    ├── StockService.java
    └── PlayerService.java
```

▼ 6. 추가도전과제1 - 입력값 검증과 전역 예외 처리

- UserRequest.java

```
package com.skala.basic.data;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class UserRequest {

    @NotBlank(message = "사용자 이름은 필수 입력 값입니다.")   // 빈 문자열 허용 안 됨
    private String username;

    @Email(message = "올바른 이메일 형식이어야 합니다.")      // 이메일 형식 검증
    private String email;

    @Size(min = 6, max = 20, message = "비밀번호는 6~20자 사이여야 합니다.")  // 길이 제한
    private String password;
}
```

  - 수정 전에는 단순히 필드와 getter/setter만 있었는데 검증 애노테이션을 붙여 잘못된 입력을 막도록 설정.

- UserController.java

```
package com.skala.basic.controller;

import com.skala.basic.data.UserRequest;
import com.skala.basic.data.UserResponse;
import jakarta.validation.Valid;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
```

```
public class UserController {

    @PostMapping
    public UserResponse createUser(@Valid @RequestBody UserRequest userRequest) {
        // 검증이 통과되면 실행되는 로직
        UserResponse response = new UserResponse();
        response.setUsername(userRequest.getUsername());
        response.setEmail(userRequest.getEmail());
        response.setMessage("사용자가 성공적으로 등록되었습니다!");
        return response;
    }
}
```

- `Valid`를 `@RequestBody UserRequest userRequest` 앞에 붙여서 DTO에 붙인 검증 애노테이션들이 실제 요청 시점에 검사

- UserResponse.java

```
package com.skala.basic.data;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserResponse {
    private String username;
    private String email;
    private String message;
}
```

- 수정 전에는 message만 있었는데 수정후에는 username과 email도 넣어서 클라이언트가확인할수있게 설정해서 컨트롤러에서 `setUsername()`이나 `setEmail()` 호출 시 오류가 나지 않게 설정

- GlobalExceptionHandler.java

```
package com.skala.basic.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    // @Valid 검증 실패 처리
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, String>> handleValidationExceptions(MethodArgumentNotVali
dException ex) {
```

```
        Map<String, String> errors = new HashMap<>();
        for (FieldError error : ex.getBindingResult().getFieldErrors()) {
            errors.put(error.getField(), error.getDefaultMessage());
        }
        return new ResponseEntity<>(errors, HttpStatus.BAD_REQUEST);
    }

    // 기타 예외 처리
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGlobalException(Exception ex) {
        return new ResponseEntity<>("서버 오류: " + ex.getMessage(), HttpStatus.INTERNAL_SERVER_ER
ROR);
    }
}
```

- 수정 전에는 일반적인 예외처리만 가능했다면 수정후에는 검증예외 MethodArgumentNotValidException를 잡아서 JSON 응답으로 바꿔주게 설정해서 잘못된 입력이 들어오면 400 응답과 함께 "username": "사용자 이름은 필수 입력 값입니다." 메시지 반환시킴.

▼ 7. 추가도전과제2 - H2 DB 데이터를 로컬 파일로 저장해서 유지하기

- application.yml

```
spring:
  datasource:
    url: jdbc:h2:file:./data/h2/skala-stock-db;DB_CLOSE_ON_EXIT=FALSE;AUTO_SERVER=TRUE
    driver-class-name: org.h2.Driver
    username: sa
    password:

  jpa:
    hibernate:
      ddl-auto: update   # 테이블 자동 생성 및 변경 반영
    show-sql: true       # 실행되는 SQL 콘솔에 출력

  h2:
    console:
      enabled: true      # /h2-console 경로에서 웹 콘솔 사용 가능
```

- `url` 을 `jdbc:h2:mem:skala-stock` (메모리 모드)에서
  `jdbc:h2:file:./data/h2/skala-stock-db;DB_CLOSE_ON_EXIT=FALSE;AUTO_SERVER=TRUE` (파일 모드)로 변경해서
  - `./data/h2/` 폴더에 `skala-stock-db.mv.db` 파일이 생성되고 앱을 껐다 켜도 데이터가 그대로 유지되게구현

▼ 8. 추가도전과제3 - OAS 기반으로 REST API 문서 자동화를 구성하기

- pom.xml

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.5.0</version>
</dependency>
```

- Swagger UI 기반 OpenAPI 문서 자동화 구현하기위해서 `springdoc-openapi` 라이브러리 추가

- UserController.java

```
package com.skala.basic.controller;

import com.skala.basic.data.UserRequest;
import com.skala.basic.data.UserResponse;
import jakarta.validation.Valid;
import org.springframework.web.bind.annotation.*;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;

@RestController
@RequestMapping("/api/users")
@Tag(name = "User API", description = "사용자 관리 API (등록, 조회 등)")
public class UserController {

    @PostMapping
    @Operation(summary = "사용자 등록", description = "신규 사용자를 등록합니다. 요청값은 @Valid 검증을 거칩니다.")
    public UserResponse createUser(@Valid @RequestBody UserRequest userRequest) {
        UserResponse response = new UserResponse();
        response.setUsername(userRequest.getUsername());
        response.setEmail(userRequest.getEmail());
        response.setMessage("사용자가 성공적으로 등록되었습니다!");
        return response;
    }

    @GetMapping("/{username}")
    @Operation(summary = "사용자 조회", description = "사용자 이름으로 등록된 사용자 정보를 조회합니다.")
    public UserResponse getUser(@PathVariable String username) {
        // 예시: 단순 조회 응답
        UserResponse response = new UserResponse();
        response.setUsername(username);
        response.setEmail(username + "@example.com");
        response.setMessage("사용자 조회 성공");
        return response;
    }
}
```

- REST API 엔드포인트만 정의되어있었는데 Swagger 어노테이션 @Tag (클래스 설명)와 @Operation (메서드 설명)을 추가해서 Swagger UI (http://localhost:8080/swagger-ui.html)에서 API 이름, 설명, 요청/응답 구조를 확인할수있게 설정.

▼ 9. 추가 API1 - 플레이어 잔액 조회 API

- PlayerController.java

```
// 플레이어 잔액 조회 API
@GetMapping("/{playerId}/balance")
public PlayerBalanceResponse getPlayerBalance(@PathVariable String playerId) {
    Double balance = playerService.getPlayerBalance(playerId);
    return new PlayerBalanceResponse(playerId, balance);
}
```

- `/api/players/{playerId}/balance` 엔드포인트를 추가해서 특정 플레이어의 현재 잔액 확인 기능 생성.

- PlayerService.java

```
// 플레이어 잔액 조회
@Transactional(readOnly = true)
public Double getPlayerBalance(String playerId) {
    Player player = playerRepository.findById(playerId)
            .orElseThrow(() -> new RuntimeException("Player not found"));
    return player.getPlayerMoney();
}
```

- 플레이어 잔액만 단독으로 반환하는 메서드 getPlayerMoney를 추가
  - 수정 전에는 잔액을 확인하려면 반드시 Player + 보유 주식 전체 정보를 가져오는 `getPlayerById`를 호출해야 했는데 플레이어 잔액만 단독으로 반환하는 메서드를 쓰면 단순히 숫자 하나만 조회하기 때문에 DB 조회 결과를 DTO 변환하는 추가 연산이 필요 없어 효율적이다.

- PlayerBalanceResponse.java

```
package com.skala.basic.dto;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class PlayerBalanceResponse {
    private String playerId;
    private Double balance;
}
```

- 잔액 조회 응답 DTO 정의

▼ 10. 추가 API2 - 플레이어 거래 기록 조회 API

- PlayerController.java

```
// 플레이어 거래 기록 조회 API
@GetMapping("/{playerId}/transactions")
public List<Transaction> getTransactions(@PathVariable String playerId) {
    return playerService.getTransactionsByPlayerId(playerId);
}
```

- `api/players/{playerId}/transactions` 엔드포인트가 추가되어, 특정 플레이어가 매수/매도한 거래 내역 확인 가능

- PlayerService.java

```
// 플레이어 거래 기록 조회
@Transactional(readOnly = true)
public List<Transaction> getTransactionsByPlayerId(String playerId) {
    return transactionRepository.findByPlayerId(playerId);
}
```

- 매수/매도 성공시 Transaction 엔티티를 생성하고 저장하도록 변경해서 getTransaction ByPlayerID()로 히스토리를 조회할수있게함

- Transaction.java

```
@Entity
@Data
@NoArgsConstructor
```

```
@AllArgsConstructor
public class Transaction {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String playerId;
    private Long stockId;
    private String stockName;
    private String type; // "BUY" or "SELL"
    private int quantity;
    private double price; // 단가
    private double totalPrice; // 총액
    private LocalDateTime timestamp;
}
```

- DB에 거래 내역을 영구 저장할수있는 새 엔티티 추가.

- TransactionRepository.java

```
public interface TransactionRepository extends JpaRepository<Transaction, Long> {
    List<Transaction> findByPlayerId(String playerId);
}
```

- 특정 플레이어 ID로 거래 기록을 조회할수있는 레퍼지토리 생성.

▼ 11. 추가 API3 - 플레이어 보유 자산 평가 API

- PlayerController.java

```
// 플레이어 보유 자산 평가 API
@GetMapping("/{playerId}/portfolio")
public PlayerPortfolioResponse getPlayerPortfolio(@PathVariable String playerId) {
    return playerService.getPlayerPortfolio(playerId);
}
```

- `/api/players/{playerId}/portfolio` 엔드포인트를 추가해서 특정 플레이어의 현금 잔액과 주식 평가액, 총 자산을 한번에확인할 수있게 생성.

- PlayerService.java

```
// 플레이어 보유 자산 평가
@Transactional(readOnly = true)
public PlayerPortfolioResponse getPlayerPortfolio(String playerId) {
    Player player = playerRepository.findById(playerId)
            .orElseThrow(() -> new RuntimeException("Player not found"));

    double balance = player.getPlayerMoney();
    double stockValue = playerStockRepository.findByPlayer_PlayerId(playerId)
            .stream()
            .mapToDouble(ps -> ps.getQuantity() * ps.getStock().getStockPrice())
            .sum();

    return new PlayerPortfolioResponse(playerId, balance, stockValue, balance + stockValue);
}
```

- `getPlayerPortfolio` 메서드로 플레이어의 보유 현금과 보유 주식 평가액을 합산해 DTO로 반환해서 자산 평가를 바로 계산

- PlayerPortfolioResponse.java

```
@Data
@AllArgsConstructor
public class PlayerPortfolioResponse {
    private String playerId;
    private Double balance;     // 보유 현금
    private Double stockValue;  // 주식 평가액
    private Double totalAsset;  // 총 자산 (balance + stockValue)
}
```

  - 자산 평가 API 전용 DTO `PlayerPortfolioResponse` 생성

▼ 12. Frontend 구현

```
frontend/
├─ Dockerfile
├─ docker-compose.yml
├─ nginx.conf
├─ .env
├─ .env.production
├─ package.json
├─ vite.config.js
├─ index.html
└─ src/
   ├─ main.js
   ├─ App.vue
   ├─ router/
   │  └─ index.js
   ├─ api/
   │  ├─ api.js
   │  ├─ hello.js
   │  ├─ users.js
   │  ├─ courses.js
   │  ├─ stocks.js
   │  └─ players.js
   └─ views/
      ├─ Dashboard.vue
      ├─ HelloView.vue
      ├─ UsersView.vue
      ├─ CoursesView.vue
      ├─ StocksView.vue
      └─ PlayersView.vue
```

- **로그인 페이지**
  - `HelloView.vue`
- **플레이어 관리 페이지**
  - `PlayersView.vue`
  - 기능
    - 플레이어 등록 (`POST /api/players`)
    - 플레이어 목록 조회 (`GET /api/players`)

        - 개별 플레이어 정보 확인 (`GET /api/players/{id}`)
        - 플레이어 주식 매수/매도 (`POST /api/players/buy` , `POST /api/players/sell`)
    - **주식 관리 페이지**
        - `StocksView.vue`
        - 기능
            - 주식 목록 조회 (`GET /api/stocks`)
            - 개별 주식 상세 조회 (`GET /api/stocks/{id}`)
            - 신규 주식 등록 (`POST /api/stocks`)
            - 주식 정보 수정 (`PUT /api/stocks`)
            - 주식 삭제 (`DELETE /api/stocks`)

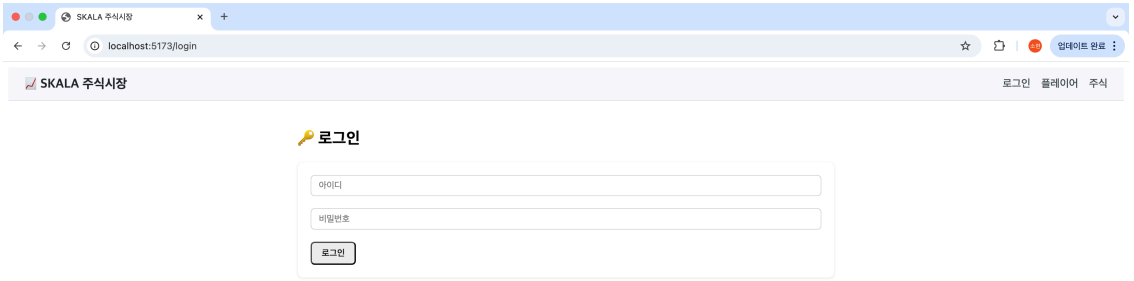
▼ 13. 백엔드 실행

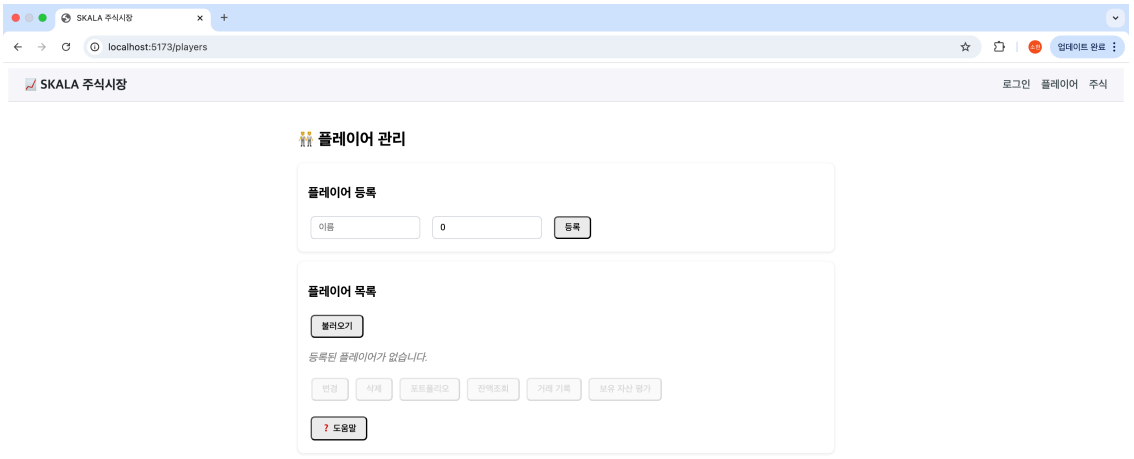
1. 실행

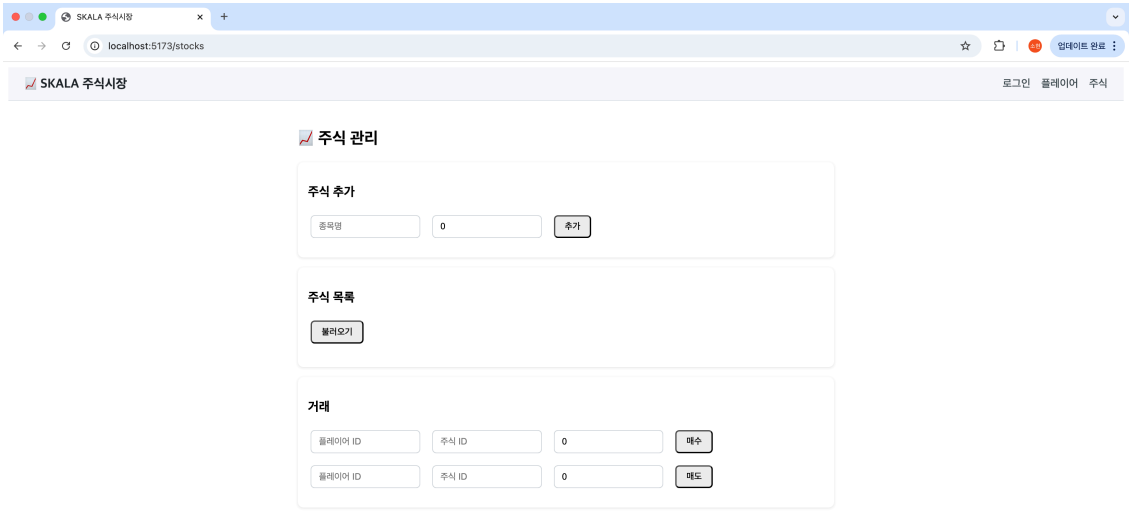
```
$ pwd
/Users/yshmbid/Documents/home/github/Spring/basic
$ mvn spring-boot:run
```

```
$ pwd
/Users/yshmbid/Documents/home/github/stock/frontend
$ npm run dev
```

2. http://localhost:5173 에서 확인







▼ 14. 컨테이너화 & 배포 (Docker + Kubernetes)

```
$ pwd
/Users/yshmbid/Documents/home/github/Spring/basic

# 도커 이미지 빌드 & 푸시
$ ./docker-build.sh
$ ./docker-push.sh

# Deployment 생성
$ kubectl apply -f k8s/deploy.yaml
```

```
# Service 생성
$ kubectl apply -f k8s/service.yaml

# YAML 다시 생성
$ user-cicd.sh -y

# 기존 리소스 삭제 후 재배포
$ kubectl delete -f k8s/deploy.yaml -n skala-practice
$ kubectl apply -f k8s/deploy.yaml -n skala-practice

# Pod 상태 확인
$ kubectl get pod -n skala-practice | grep sk019 # Running으로떠야 정상
sk019-myfirst-api-server-58dff9f96b-jx5n6   1/1     Running            0            19m # Pending 상태

# 포트포워딩으로 API 접근
$ kubectl port-forward svc/sk019-myfirst-api-server 8080:8080 -n skala-practice
Forwarding from 127.0.0.1:8080 → 8080
Forwarding from [::1]:8080 → 8080

# curl로 확인
$ curl http://localhost:8080/hello

# 브라우저로 확인
https://ingress.skala25a.project.skala-ai.com/sk019-api/redoc.html 링크로 접근
```

- 실행결과
  - Pod 상태 확인시 Running으로 뜨지 않고 Pending으로 뜨고 있어 `kubectl get ingress -n skala-practice` 상에서 확인되지않는상태

```
$ kubectl get ingress -n skala-practice
NAME                     CLASS         HOSTS                                ADDRESS   PORTS     AGE
backend-test-ingress     public-nginx  backend.skala25a.project.skala-ai.com           80, 443   3d20h
echo                     public        ingress.skala25a.project.skala-ai.com          80        8h
echo2                    public        ingress.skala25a.project.skala-ai.com          80        8h
sk007-ing                nginx         ingress.skala25a.project.skala-ai.com          80        9h
sk018-skala-stock        public-nginx  ingress.skala25a.project.skala-ai.com          80, 443   9h
sk024-ingress            public-nginx  ingress.skala25a.project.skala-ai.com          80, 443   11h
sk025-skala-stock-api    public-nginx  ingress.skala25a.project.skala-ai.com          80, 443   9h
sk025-skala-stock-frontend  public-nginx  ingress.skala25a.project.skala-ai.com       80, 443   9h
sk029-skala-stock-ingress   public-nginx  ingress.skala25a.project.skala-ai.com       80        1h
```

    - Pod이 Running 상태로만 되면 `kubectl get ingress -n skala-practice` 상에서 확인 가능.