DOCUMENTATION FOR SMART SUPPLIER PORTAL

WEEK 1:

Alright—let's execute **W1** in order:

1.1 DB Migration

Path: `api/src/main/resources/db/migration/V1__create_order.sql`

Create the folder:

```
mkdir -p api/src/main/resources/db/migration
```

Create the migration file:

```
cat > api/src/main/resources/db/migration/V1__create_order.sql ;
```

```
CREATE TABLE orders (
        id SERIAL PRIMARY KEY,
        supplier VARCHAR(255) NOT NULL,
        amount DECIMAL(10,2) NOT NULL,
        created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Run: Start your Spring Boot app; Flyway will auto-pick this up and create the table.

1.2 JPA Entity & Repository

Order.java

Path: `api/src/main/java/com/portal/model/Order.java`

OrderRepository.java

Path: `api/src/main/java/com/portal/repository/OrderRepository.java`

1.3 REST Controller

Path: `api/src/main/java/com/portal/controller/OrderController.java`

1.4 OpenAPI

Add dependency in `api/pom.xml` within `<dependencies>`:

```
 <dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.1.0</version>
</dependency>
```

1.
2. **Restart** your app and confirm:

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

1.5 Generate TS Client (npm)

Install the generator in `web/`:

```
 cd web
npm install --save-dev openapi-typescript-codegen
```

1. **Add script** to `web/package.json`:

```
          "scripts": {
 "openapi": "openapi --input http://localhost:8080/v3/api-docs --output
web/src/api --client axios --exportServices true"
}
```

2. **Run** the codegen:
   npm run openapi

3. You should now see typed files under `web/src/api`.

1.6 React Pages

1.6.1 Orders List Page

Path: `web/app/orders/page.tsx`

1.6.2 New Order Form Page

Path: `web/app/orders/new/page.tsx`

1.7 Form Validation

Install:

```
npm install react-hook-form zod @hookform/resolvers
```

Define schema in `web/src/api/models.ts`:

1.8 React Query Hook

Install:

```
npm install @tanstack/react-query
```

Create `web/src/hooks/useOrders.ts`:

Wrap your app in `QueryClientProvider` (e.g., in `web/app/layout.tsx`).

1.9 Styling (OPTIONAL)

Tailwind CSS:

```
 npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Configure `tailwind.config.js`:

```
module.exports = {
 content: ['./app/**/*.{ts,tsx}'],
 theme: { extend: {} },
 plugins: [],
};
```

shadcn/ui:

1. npm install @shadcn/ui lucide-react
2. **Apply** `<Card>`, `<Button>`, and utility classes in your pages as demonstrated above.

1.10 Slice Docs

Path: `docs/slice-1.md`

```
# Slice 1: Orders CRUD

## Overview
Implement end-to-end “orders” feature:
- DB migration
- JPA entity & repo
- REST API (`POST`, `GET`)
- OpenAPI docs
- Generated TS client
- React pages with validation & styling
- Slice documentation

## Migration
`V1__create_order.sql` creates `orders` table.

## Backend
- `Order.java`, `OrderRepository.java`
- `OrderController.java` → `POST /api/orders`, `GET /api/orders`

## API Docs
- Swagger UI: `/swagger-ui.html`
- OpenAPI JSON: `/v3/api-docs`

## Frontend
- Generated client: `web/src/api`
- List: `/orders`
```

~~~
- New form: `/orders/new`
- Validation: react-hook-form + zod
- Data: React Query

## Styling
- Tailwind CSS + shadcn/ui components

## Validation
```bash
curl -X POST localhost:8080/api/orders \
  -H "Content-Type: application/json" \
  -d '{"supplier":"ACME","amount":123.45}'


---

### ✅ Final Validation

```bash
curl -X POST localhost:8080/api/orders \
  -H "Content-Type: application/json" \
  -d '{"supplier":"ACME","amount":123.45}'
# Expect JSON { id, supplier, amount, createdAt }

curl localhost:8080/api/orders
# Expect array with at least the above order
~~~

—---

SOME KEY LESSONS WHEN DEALING WITH CLASSES OR CONTROLLERS IN SBOOT:

Here are several ways you can wire up the exact same two endpoints—each using a different combination of Spring annotations or injection style. Pick whatever fits your taste or team conventions:

---

1) “Modern” constructor-injection + composed shortcuts

```java
@RestController
@RequestMapping("/api/orders")
public class OrderController {
    private final OrderRepository repo;
    public OrderController(OrderRepository repo) {
        this.repo = repo;
    }

    @PostMapping
```

```
    public Order create(@RequestBody Order order) {
        return repo.save(order);
    }

    @GetMapping
    public List<Order> list() {
        return repo.findAll();
    }
}
```

- **No** `@Autowired` needed on the lone constructor
- Uses the new, concise `@PostMapping`/`@GetMapping`
- Handler methods **must** be `public` so Spring can detect & invoke them

---

2) `@Controller` + `@ResponseBody` + classic `@RequestMapping`

```
@Controller
@RequestMapping("/api/orders")
public class OrderController {
    private final OrderRepository repo;
    @Autowired       // explicit field-injection
    public OrderController(OrderRepository repo) {
        this.repo = repo;
    }

    @RequestMapping(method = RequestMethod.POST,
                    consumes = MediaType.APPLICATION_JSON_VALUE,
                    produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public Order create(@RequestBody Order input) {
        return repo.save(input);
    }

    @RequestMapping(method = RequestMethod.GET,
                    produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public List<Order> list() {
        return repo.findAll();
    }
}
```

- Uses the older, fully-qualified `@RequestMapping` with `method=`
- `@Controller` + `@ResponseBody` is functionally the same as `@RestController`

---

3) Field-injection + shortcut mappings

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {
   @Autowired
   private OrderRepository repo;    // field-injected

   @PostMapping("/")
   public Order create(@RequestBody Order order) {
      return repo.save(order);
   }

   @GetMapping("/")
   public List<Order> list() {
      return repo.findAll();
   }
}
```

- **Not recommended** for testability, but very terse
- Explicit trailing slash in the path—sometimes helps with consistency

---

4) Setter-injection + composed shortcuts

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {
   private OrderRepository repo;

   @Autowired
   public void setOrderRepository(OrderRepository repo) {
      this.repo = repo;
   }

   @PostMapping
   public Order create(@RequestBody Order order) {
      return repo.save(order);
```

```
    }

    @GetMapping
    public List<Order> list() {
        return repo.findAll();
    }
}
```

- Allows “late” injection if you have circular deps (rare)
- Setter can also be `protected` or `public`

---

5) Lombok + explicit status codes

```
@RestController
@RequestMapping("/api/orders")
@RequiredArgsConstructor            // generates the constructor for your final fields
public class OrderController {
    private final OrderRepository repo;

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Order create(@RequestBody @Valid Order order) {
        return repo.save(order);
    }

    @GetMapping
    public ResponseEntity<List<Order>> list() {
        return ResponseEntity.ok(repo.findAll());
    }
}
```

- `@RequiredArgsConstructor` from Lombok replaces boilerplate
- `@ResponseStatus` or `ResponseEntity` for fine-grained HTTP codes

---

6) Global CORS + OpenAPI grouping

```
@CrossOrigin(origins = "*")
@Tag(name = "Orders", description = "CRUD operations on orders")
@RestController
```

```
@RequestMapping("/api/orders")
public class OrderController {
   private final OrderRepository repo;
   public OrderController(OrderRepository repo) { this.repo = repo; }

   @Operation(summary = "Create a new order")
   @PostMapping
   public Order create(@RequestBody Order order) {
      return repo.save(order);
   }

   @Operation(summary = "List all orders")
   @GetMapping
   public List<Order> list() {
      return repo.findAll();
   }
}
```

- Adds `@CrossOrigin` for browser clients
- Uses Swagger Annotations (`@Tag`, `@Operation`) to enrich your OpenAPI docs

---

Key takeaways:

- **Constructor injection** (no `@Autowired` on a single constructor) is the cleanest, most testable approach.
- Handler methods **must** be `public`.
- You can trade off brevity, explicitness, and extra features (status codes, CORS, docs) by choosing different annotations.

Key Commands to connect to DB:

Backend changes:

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/portal
spring.datasource.username=portal_user
spring.datasource.password=your_password
spring.jpa.hibernate.ddl-auto=update
```

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

And add dependency,

```
<dependency>
   <groupId>org.postgresql</groupId>
   <artifactId>postgresql</artifactId>
   <version>42.3.1</version>
</dependency>
```

DB/terminal based changes:
Run →

- sudo apt update
- sudo apt install postgresql postgresql-contrib

1. **sudo systemctl start postgresql**
2. **sudo -i -u postgres**
3. **psql**
4. **CREATE DATABASE portal;**
5. **CREATE USER portal_user WITH PASSWORD 'portal_pass';**
6. **GRANT ALL PRIVILEGES ON DATABASE portal TO portal_user;**
7. **psql -U portal_user -d portal -h localhost and you should be logged in now run the app and table in portal DB should be visible**

RECAP:

Slice 1: Orders CRUD

Overview

This slice delivers a minimal end-to-end "orders" feature, covering:

- **Database migration (Flyway)**
- **JPA entity & Spring Data repository**
- **REST API (`POST` + `GET`)**

- OpenAPI documentation & Swagger UI
- TypeScript client generation
- React pages for listing & creating orders
- Form validation with React Hook Form + Zod
- Data fetching with React Query
- Styling with Material UI

1. Database Migration

Location: `api/src/main/resources/db/migration/V1__create_order.sql`

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  supplier VARCHAR(255) NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

2. Backend

2.1 JPA Entity

File: `api/src/main/java/com/portal/model/Order.java`

```
@Entity
@Table(name = "orders")
public class Order {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String supplier;
  private BigDecimal amount;
  @Column(name = "created_at", updatable = false)
  private Instant createdAt = Instant.now();
  // getters/setters omitted
}
```

2.2 Repository

File: `api/src/main/java/com/portal/repository/OrderRepository.java`

```
@Repository
public interface OrderRepository extends JpaRepository<Order, Long> {}
```

2.3 REST Controller

File: `api/src/main/java/com/portal/controller/OrderController.java`

```
@CrossOrigin(origins = "http://localhost:3000")
@RestController
@RequestMapping("/api/orders")
public class OrderController {
  private final OrderRepository repo;
  public OrderController(OrderRepository repo) { this.repo = repo; }

  @PostMapping
  public Order create(@RequestBody Order order) { return repo.save(order); }

  @GetMapping
  public List<Order> list() { return repo.findAll(); }
}
```

3. OpenAPI & Swagger UI

Dependency: add to `pom.xml`

`>This does all the work`

```
 <dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.1.0</version>
</dependency>
```

Access:

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

`Some EXTRA Steps in order to manually update configuration for swagger and open API and some information about it:`

The /v3/api-docs endpoint is automatically provisioned at runtime by the springdoc-openapi library you added to your Spring Boot app. Here's how it works under the hood:

Dependency Activation
When you include

```
 <dependency>

  <groupId>org.springdoc</groupId>

  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

  <version>2.1.0</version>

</dependency>
```

1. Spring Boot's auto-configuration picks up the starter and registers all the beans needed to generate an OpenAPI description and serve the Swagger UI.

2. Controller & Model Scanning
   At application startup, springdoc scans your application context for:

   - @RestController (and @Controller) request mappings

   - Request/response bodies, model classes (@Schema, Jackson annotations, etc.)

   - Swagger/OpenAPI annotations if you've added any (@Operation, @Parameter, @Schema, etc.)

3. Building the OpenAPI Model
   springdoc uses that metadata to build an in-memory OpenAPI object (following the OpenAPI 3.0 spec). This includes:

   - paths (your @GetMapping, @PostMapping endpoints)

   - components (schemas for your DTOs/entities)

    - Security schemes, servers, tags, etc., all gleaned from your code or defaults.

4. Exposing the JSON
   It then registers a handler at GET /v3/api-docs that, when invoked, serializes that OpenAPI object to JSON. That's the same JSON that Swagger UI (and your codegen script) consume.

5. Serving Swagger UI
   Alongside, the -starter-webmvc-ui dependency also wires up a static Swagger UI under GET /swagger-ui.html (and its associated JS/CSS). That UI fetches /v3/api-docs to render the interactive docs.

Customization

You can tweak the paths or behavior via properties in application.properties or application.yml:

```
# Change the JSON endpoint

springdoc.api-docs.path=/api-docs


# Change the Swagger UI path

springdoc.swagger-ui.path=/swagger-ui.html


# Limit the packages to scan

springdoc.packagesToScan=com.portal.controller,com.portal.model
```

But out of the box, no manual controller or JSON file is required—springdoc does it all dynamically at startup.

4. TypeScript Client Generation

In `web/package.json`:

```
"scripts": {
  "openapi": "npx openapi-typescript-codegen --input http://localhost:8080/v3/api-docs
--output src/api --client axios --exportServices true"
}
```

Run:

```
cd web
npm run openapi
```

Generated in `web/src/api/core`, `models/`, and `services/OrderControllerService.ts`.

5. Frontend

5.1 Zod Schemas & Types

File: `web/src/api/models.ts`

```
import { z } from 'zod';
export const orderInputSchema = z.object({
  supplier: z.string().min(1),
  amount: z.number().positive(),
});
export type OrderInput = z.infer<typeof orderInputSchema>;
export const orderSchema = orderInputSchema.extend({
  id: z.number(),
  createdAt: z.string(),
});
export type Order = z.infer<typeof orderSchema>;
```

5.2 React Query Hook

File: `web/src/hooks/useOrders.ts`

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { OrderControllerService } from '@/api/services/OrderControllerService';
import type { OrderInput } from '@/api/models';

export function useOrders() {
  const client = useQueryClient();
  const listQuery = useQuery(['orders'], () => OrderControllerService.list());
  const createMutation = useMutation(
    (input: OrderInput) => OrderControllerService.create(input as any),
    { onSuccess: () => client.invalidateQueries(['orders']) }
  );
  return { ...listQuery, orders: listQuery.data ?? [], createOrder: createMutation };
}
```

5.3 React Pages

- List Orders: `web/app/orders/page.tsx` (uses `useOrders`)
- New Order Form: `web/app/orders/new/page.tsx` (react-hook-form + MUI)

6. Validation & Styling

- Validation: react-hook-form + Zod (`zodResolver`)
- Styling: Material UI ThemeProvider + components (Paper, TextField, Button)

7. Manual Validation

```
# Create
curl -X POST http://localhost:8080/api/orders \
 -H "Content-Type: application/json" \
 -d '{"supplier":"ACME","amount":123.45}'
# List
curl http://localhost:8080/api/orders
```

Keep this doc updated as you extend the feature in subsequent slices. Feel free to add screenshots or code snippets as needed!