Exception Handling

Benefits of Exception Handling

- Prevent application crashes
- Provide user-friendly error responses
- Facilitate debugging and maintenance
- Ensure consistent error handling across the application

Handling Exceptions

- Use `@ExceptionHandler` to handle specific exceptions in controllers
- Use `@RestControllerAdvice` for global exception handling
- Return appropriate HTTP status codes and error messages
- Use Custom error response class to provide structured error details

Exception Handlers at Controller Level

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    @GetMapping(path = "/{employeeId}")
    public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariab
le Long employeeId) {
      Optional<EmployeeDTO> employeeDTO = employeeService.getEmp
loyeeById(employeeId);

     return employeeDTO
             .map(employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
         .orElseThrow(() -> new NoSuchElementException(
             "Employee with id " + employeeId + " not found")
           );
    }

    @ExceptionHandler(NoSuchElementException.class)
    public ResponseEntity<String> handleEmployeeNotFound(NoSuchEle
mentException exception) {
        return new ResponseEntity<>("Employee not found", HttpStatus.NO
T_FOUND);
    }
}
```

There is another better way to keep the controller code clean is to move these to a global-level exception handler that takes care of all such logics.

Mark it to be more of a generic error such as Resource:

```
@GetMapping(path = "/{employeeId}")
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable Lo
ng employeeId) {
  Optional<EmployeeDTO> employeeDTO = employeeService.getEmploye
eById(employeeId);

 return employeeDTO
          .map(employeeDTO1 → ResponseEntity.ok(employeeDTO1))
      .orElseThrow(() → new ResourceNotFoundException("Employee with
id " + employeeId + " not found"));
}
```

A very simple class that extends the RuntimeException class:

```
public class ResourceNotFoundException extends RuntimeException {
  public ResourceNotFoundException(String message) {
    super(message);
  }
}
```

To maintain a standard of how we response back to the user and keep it more predictable response lets declare a ApiError class that gives us a structured response to send:

```
@Data
@Builder
public class ApiError {
    private HttpStatus status;
    private String message;
    private List<String> subErrors;
}
```

Now on the meat part of this entire discussion is the have a custom GlobalExceptionHandler that can hold multiple use-cases of exceptions that may go wrong with run-time of the application.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ApiError> handleResourceNotFound(ResourceNo
tFoundException exception) {
        ApiError apiError = ApiError
                        .builder()
                        .status(HttpStatus.NOT_FOUND)
                        .message(exception.getMessage())
                        .build();

        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }
}
```

Using the builder design pattern we construct the api response and send response back. To handle more such we have these also that can tag along:

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ApiError> handleInputValidationErrors(MethodArgu
mentNotValidException exception) {
        List<String> errors = exception
                                        .getBindingResult()
                                        .getAllErrors()
                                        .stream()
                                        .map(err → err.getDefaultMessage())
                                        .collect(Collectors.toList());

    ApiError apiError = ApiError
                                    .builder()
                                    .status(HttpStatus.BAD_REQUEST)
                                    .message("Input validation failed")
                                    .subErrors(errors)
                                    .build();

    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
}

@ExceptionHandler(Exception.class)
public ResponseEntity<ApiError> handleException(Exception exception) {
       ApiError apiError = ApiError
                                               .builder()
                           .status(HttpStatus.INTERNAL_SERVER_ERROR)
                           .message(exception.getMessage())
                           .build();

     return new ResponseEntity<>(apiError, HttpStatus.INTERNAL_SERVER_E
RROR);
}
```