Routine App Tutorial Stage 1: Initial Prototype (Monolithic) - Detailed Instructions

This document outlines the structure and implementation of the dev.algo.routine full stack application. Specifically, this document addresses how the monolithic application is built.

1. Backend Setup

1.1 Set up Spring Boot project

1. Visit https://start.spring.io/
2. Configure: Spring Initializr Configuration
3. Click "Generate", download, and unzip
4. In IDEA, import the project: Project properties > Import module

1.2 Configure PostgreSQL database

Setting up a robust database is crucial for our application. We'll use PostgreSQL, a powerful open-source relational database system.

1. Install PostgreSQL if not already installed
2. Create a new database named `routine_app`
3. Open `src/main/resources/application.properties` and add:

```
spring.application.name=backend

# DB connection
spring.datasource.url=jdbc:postgresql://localhost:5432/routine_db
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true

# The following line is for using Spring Profiles
 spring.profiles.active=dev
```

Create a file `application-dev.properties` and put the content below. Make sure to gitignore the file, to avoid committing user:pass on git.

```
spring.datasource.username=postgres
spring.datasource.password=!DB$dmin
```

NOTE In a production environment, consider using environment variables or a secure vault for sensitive information.

1.3 Create JPA entities

Entities are the backbone of our data model. They represent the structure of our database tables in Java objects.

1. Create a new package `dev.algo.routine.backend.model`
2. Create `User.java`:

```java
package dev.algo.routine.backend.model;

import jakarta.persistence.*;
import lombok.Data;

@Data
@Entity
@Table(name = "users")
public class User {

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

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false, unique = true)
    private String email;
}
```

1. Create `Task.java`:

```java
package dev.algo.routine.backend.model;

import jakarta.persistence.*;
import lombok.Data;
import java.time.LocalDateTime;

@Data
@Entity
@Table(name = "tasks")
public class Task {

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

    @Column(nullable = false)
    private String title;

    private String description;

    @Column(nullable = false)
    private LocalDateTime dueDate;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    private boolean complete;
}
```

NOTE Consider adding additional fields like `createdAt` and `updatedAt` for better tracking of records.

1.4 Implement repositories

Repositories provide an abstraction layer for database operations, allowing us to interact with our entities easily.

1. Create a new package `dev.algo.routine.backend.repository`
2. Create `UserRepository.java`:

```java
package dev.algo.routine.backend.repository;

import dev.algo.routine.backend.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username);
}
```

1. Create TaskRepository.java:

```
package dev.algo.routine.backend.repository;

import dev.algo.routine.backend.model.Task;
import dev.algo.routine.backend.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

public interface TaskRepository extends JpaRepository<Task, Long> {
    List<Task> findByUser(User user);
}
```

NOTE For more complex queries, consider using @Query annotations or QueryDSL for type-safe queries.

1.5 Create services

Services encapsulate our business logic, providing a clean separation between the web layer and data access layer.

1. Create a new package dev.algo.routine.backend.service
2. Create UserService.java:

```
package dev.algo.routine.backend.service;

import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.repository.UserRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    private static Logger logger = LoggerFactory.getLogger(UserService.class);

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public UserService(UserRepository userRepository, PasswordEncoder passwordEncoder)
{
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    public User createUser(User user) {
        // encode the password before saving
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        return userRepository.save(user);
    }

    public User findByUsername(String username) {
        return userRepository.findByUsername(username);
    }
}
```

1. Create `TaskService.java`:

```
package dev.algo.routine.backend.service;

import dev.algo.routine.backend.model.Task;
import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.repository.TaskRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class TaskService {

    private final TaskRepository taskRepository;

    public TaskService(TaskRepository taskRepository) {
        this.taskRepository = taskRepository;
    }

    public Task createTask(Task task) {
//         TODO: a deeper check might be needed
//         if (task.getUser() == null){
//             throw new IllegalArgumentException("Task must be associated with a
user");
//         }
//         if (task.getDeadline()==null){
//             task.setDeadline(LocalDateTime.now().plusDays(1));//default due date
tomorrow
//         }
        return taskRepository.save(task);
    }

    public List<Task> getTaskForUser(User user){
        return taskRepository.findByUser(user);
    }

    public Task updateTask(Task task) {
        return taskRepository.save(task);
    }
    public void deleteTask(Long taskId) {
        taskRepository.deleteById(taskId);
    }
}
```

NOTE Consider adding validation logic and error handling in these service methods for robustness.

1.6 Implement REST controllers

Controllers handle HTTP requests and responses, defining the API endpoints for our application.

1. Create a new package `dev.algo.routine.backend.controller`
2. Create `UserController.java`:

```java
package dev.algo.routine.backend.controller;

import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

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

    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }

    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        User createdUser = userService.createUser(user);
        return ResponseEntity.ok(createdUser);
    }
}
```

1. Create `TaskController.java`:

```java
package dev.algo.routine.backend.controller;

```

```java
import dev.algo.routine.backend.model.Task;
import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.service.TaskService;
import dev.algo.routine.backend.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/tasks")
public class TaskController {

    private final TaskService taskService;
    private final UserService userService;

    public TaskController(TaskService taskService, UserService userService) {
        this.taskService = taskService;
        this.userService = userService;
    }

    @PostMapping
    public ResponseEntity<Task> createTask(@RequestBody Task task, Authentication
authentication) {
        // Get the authenticated user
        User user = userService.findByUsername(authentication.getName());
        task.setUser(user);
        Task createdTask = taskService.createTask(task);
        return ResponseEntity.ok(createdTask);
    }

    @GetMapping
    public ResponseEntity<List<Task>> getTasks(Authentication authentication) {
        User user = userService.findByUsername(authentication.getName());
        List<Task> tasks = taskService.getTaskForUser(user);
        return ResponseEntity.ok(tasks);
    }

    @PutMapping("/{taskId}")
    public ResponseEntity<Void> deleteTask(@PathVariable Long taskId) {
        taskService.deleteTask(taskId);
        return ResponseEntity.ok().build();
    }
}
```

NOTE Consider implementing pagination for the getTasks endpoint to handle large numbers of tasks efficiently.

1.7 Implement basic Spring Security configuration

Security is crucial for any application. Here, we set up basic authentication and authorization rules.

1. Create a new package `dev.algo.routine.backend.config`
2. Create `SecurityConfig.java`:

```java
package dev.algo.routine.backend.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.Authentica
tionConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.Arrays;

@Configuration
@Profile({"dev", "test"})
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http
                .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for simplicity.
Enable in production.
                .cors( cors -> cors.configurationSource(corsConfigurationSource()))
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/api/users/register", "/api/auth/login")
.permitAll()
                        .requestMatchers("/api/tasks/**").authenticated()
                        .anyRequest().authenticated()
                )
                .httpBasic( httpBasic -> {}); // Use HTTP Basic Authentication
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("http://localhost:3000"));//
allow origin: react dev server
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE",
"OPTIONS"));
        configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"
));
        configuration.setExposedHeaders(Arrays.asList("Authorization"));// headers the
browser can access
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L);// how long browser should cache CORS
configuration (seconds)
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(
);
        source.registerCorsConfiguration("/**", configuration); // CORS config applied
to all paths
        return source;
    }
}
```

1. Create `CustomUserDetailsService.java`:

```
package dev.algo.routine.backend.config;

import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.service.UserService;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserService userService;
    public CustomUserDetailsService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userService.findByUsername(username);
        if(user == null) {
            throw new UsernameNotFoundException("User not found with username: " +
username);
        }
        // Convert our custom User to Spring's UserDetails
        return org.springframework.security.core.userdetails.User
                .withUsername(user.getUsername())
                .password(user.getPassword())
                .roles("USER")
                .build();
    }
}
```

NOTE For production, consider implementing JWT (JSON Web Tokens) for stateless authentication and more granular authorization rules.

1.8 Add Login Endpoint

Implement an `/api/auth/login` endpoint, to allow login from the frontend

1. Create `AuthController.java`

```
package dev.algo.routine.backend.controller;

import dev.algo.routine.backend.model.User;
import dev.algo.routine.backend.service.UserService;
import lombok.Data;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
```

```java
import org.springframework.web.bind.annotation.RestController;

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

@RestController
@RequestMapping("/api/auth")
public class AuthController {
    private static final Logger logger = LoggerFactory.getLogger(AuthController.class
);

    private AuthenticationManager authenticationManager;
    private UserService userService;

    public AuthController(AuthenticationManager authenticationManager, UserService
userService) {
        this.authenticationManager = authenticationManager;
        this.userService = userService;
    }

    @PostMapping("/login")
    public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest loginRequest)
{
        logger.info("Login attempt for user: " + loginRequest.getUsername());
        try{
            Authentication authentication = authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            loginRequest.getUsername(),
                            loginRequest.getPassword()
                    )
            );

            SecurityContextHolder.getContext().setAuthentication(authentication);

            User user = userService.findByUsername(loginRequest.getUsername());
            logger.info("User authenticated successfully: {}", user.getUsername());

            Map<String, String> response = new HashMap<>();
            response.put("token", "dummy-token");// Replace with JWT in production
            response.put("username", user.getUsername());

            return ResponseEntity.ok(response);
        } catch (AuthenticationException e) {
            logger.error("Authentication failed for user: " + loginRequest.
getUsername());
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                    .body("Invalid username or password");
        }
    }
}
```

```
@Data
class LoginRequest {
    private String username;
    private String password;
}
```

1. Update `SecurityConfig.java` to configure CORS, enable unauthenticated access to login endpoint, and authenticated access to the tasks endpoints

```
package dev.algo.routine.backend.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.Authentica
tionConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.Arrays;

@Configuration
@Profile({"dev", "test"})
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
        http
                .csrf(AbstractHttpConfigurer::disable) // Disable CSRF for simplicity.
Enable in production.
                .cors( cors -> cors.configurationSource(corsConfigurationSource()))
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/api/users/register", "/api/auth/login")
.permitAll()
                        .requestMatchers("/api/tasks/**").authenticated()
                        .anyRequest().authenticated()
                )
                .httpBasic( httpBasic -> {}); // Use HTTP Basic Authentication
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("http://localhost:3000"));//
allow origin: react dev server
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE",
"OPTIONS"));
        configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"
));
        configuration.setExposedHeaders(Arrays.asList("Authorization"));// headers the
browser can access
        configuration.setAllowCredentials(true);
        configuration.setMaxAge(3600L);// how long browser should cache CORS
configuration (seconds)
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(
);
        source.registerCorsConfiguration("/**", configuration); // CORS config applied
to all paths
        return source;
    }
}
```

1.9 Running the Backend

To run the application:

1. Ensure PostgreSQL is running and the database is created
2. Run the Spring Boot application
3. The API will be available at `http://localhost:8080`

```
mvn clean install -U
mvn spring-boot:run
```

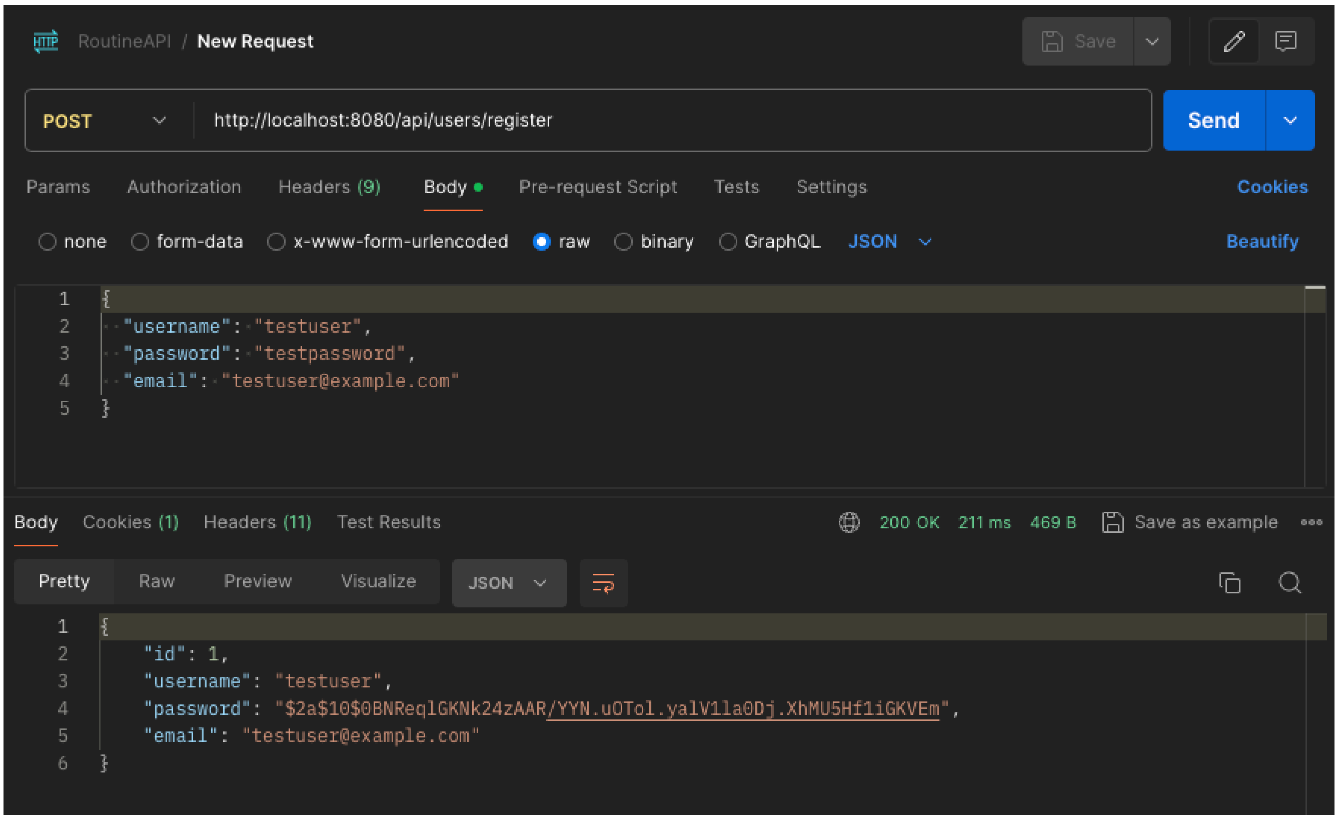
1.10 API Endpoints

When the backend is running, you can test the endpoints with a tool like Postman:

- POST `/api/users/register`: Register a new user
- POST `/api/tasks`: Create a new task
- GET `/api/tasks`: Get all tasks for the authenticated user
- PUT `/api/tasks/{id}`: Update a task
- DELETE `/api/tasks/{id}`: Delete a task

Example: Use Postman to send a POST request to http://localhost:8080/api/users/register with a JSON body like this:

```json
{
  "username": "testuser",
  "password": "testpassword",
  "email": "testuser@example.com"
}
```



```
{
  "username": "testuser",
  "password": "testpassword",
  "email": "testuser@example.com"
}
```

```
{
    "id": 1,
    "username": "testuser",
    "password": "$2a$10$0BNReqlGKNk24zAAR/YYN.uOTol.yalV1la0Dj.XhMU5Hf1iGKVEm",
    "email": "testuser@example.com"
}
```

If the user registration is successful, you should be able to use these credentials to log in at the browser prompt. So, access https://localhost8080 and login with testuser:testpassword

1.11 Notes

Remember to implement proper error handling, validation, and testing for a production-ready application.