Spring Boot Multi-tenant Architecture Overview

Kostiantyn Dementiev · Follow
8 min read · Dec 26, 2022

Developing and maintaining Spring Boot multi-tenant applications can land lots of developers in great trouble.

This article is about creating the Spring Boot multi-tenant project (using database-per-tenant approach) with the ability to add new databases in application runtime.

As a demonstration of the project with high security requirments medical lab simulation will be created. Technology stack: **Spring Boot** + **Spring Data JPA** & **Spring Data JDBC**, for test purposes **Testcontainers** + **Database Rider** + **p6spy** will be used.

Table of Contents

- Database workflow
  - Dynamic database creation
  - Dynamic database choosing
  > Spring Data JPA
  > Spring Data JDBC
- Integration tests
- Summary

Multi-tenancy concept overview

The *multi-tenancy* principle allows several users to share computing, networking, and storage resources without ever having access to one another's data. Each client (referred as *tenant*) could receive a customized version of a multi-tenant application, but its overall architecture and core features remain the same. Multi-tenancy is a tactic that Software-as-a-Service (SaaS) providers frequently employ.

Lexicon

- Tenant — client, served by one or multiple application instances
- Single-tenant application — type of a solution, where all application instances are serving only one tenant
- Multi-tenant application — type of a solution, where application instances are serving 2 or more tenants

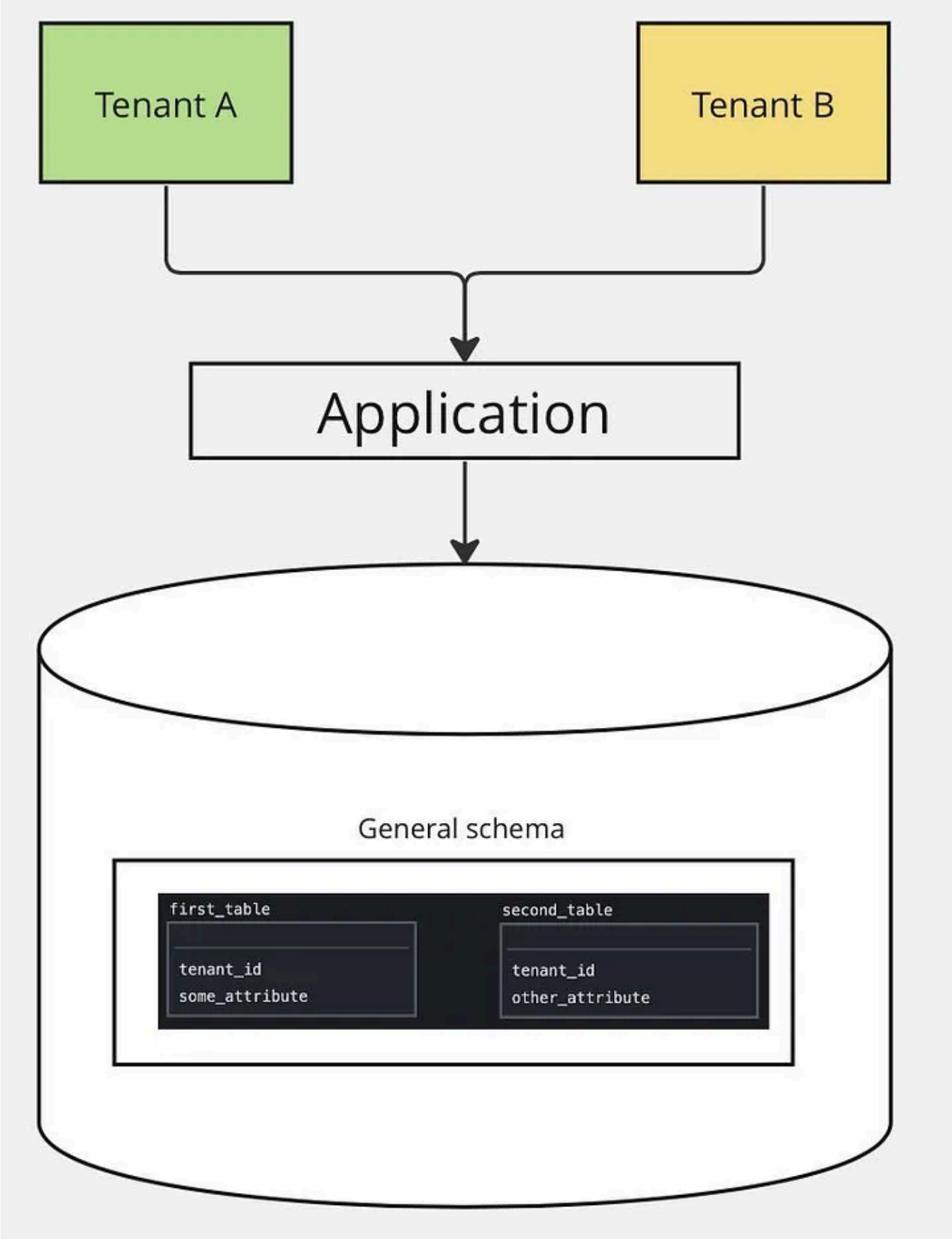
Multi-tenancy implementation approaches

In general there are 3 most commonly used types of multi tenant architecture implementation:

- Shared schema
- Schema-per-tenant
- Database-per-tenant

Shared schema

This is the most popular approach. Using it, the data from all tenants is stored in one common schema. To separate data from different tenants in some tables we can specify the column which will identify the tenant:



Advantages:

- Best development simplicity (this is well-known approach for most of the developers, used in the majority of applications)

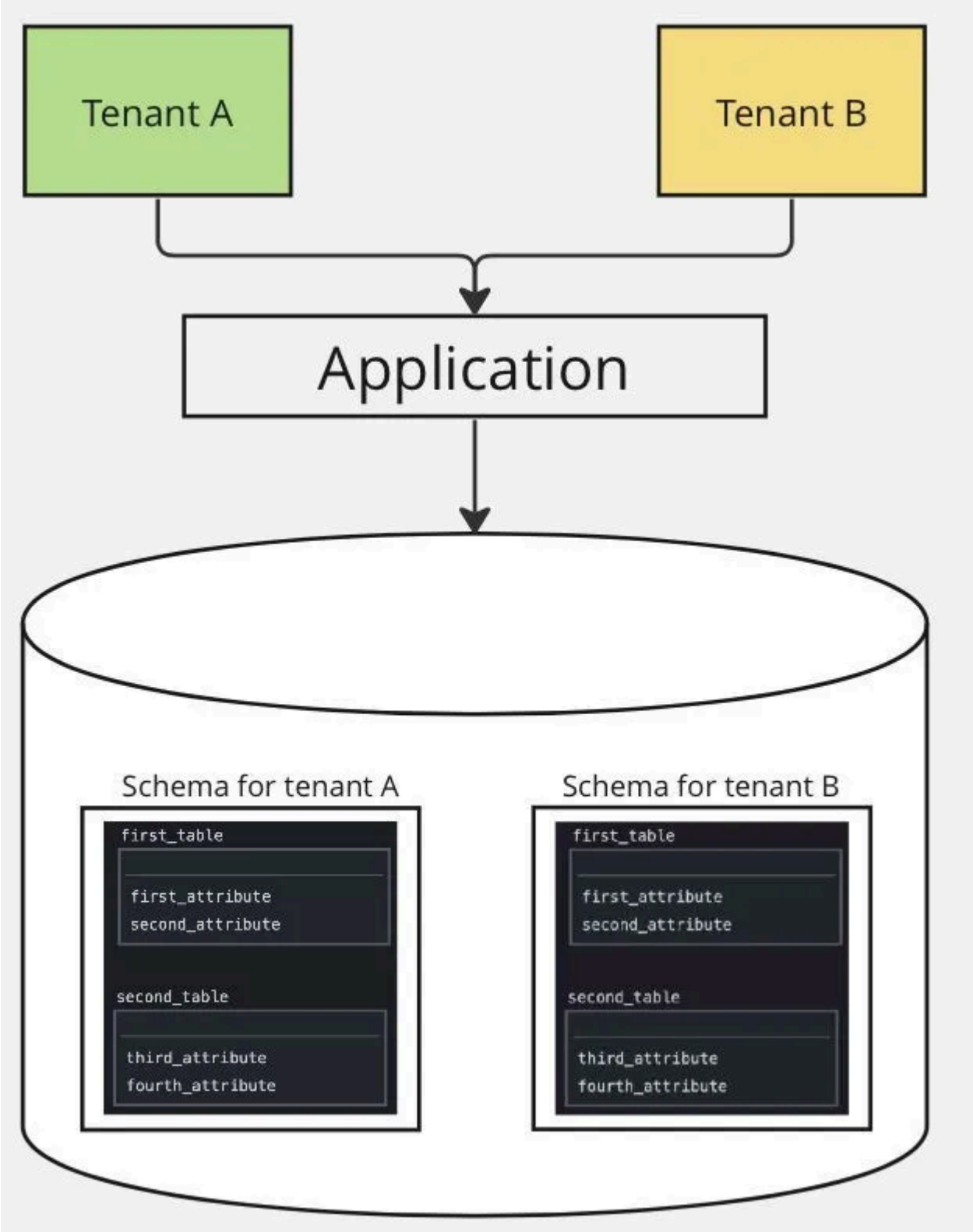
- Low infrastructure cost (we have to pay only for one database instance)

Disadvantages:

- Worse database performance (because a lot of data is stored in same tables)

Schema-per-tenant

Using this approach, we can store tenant's data in separate schemas on shared database instance:



Advantages:

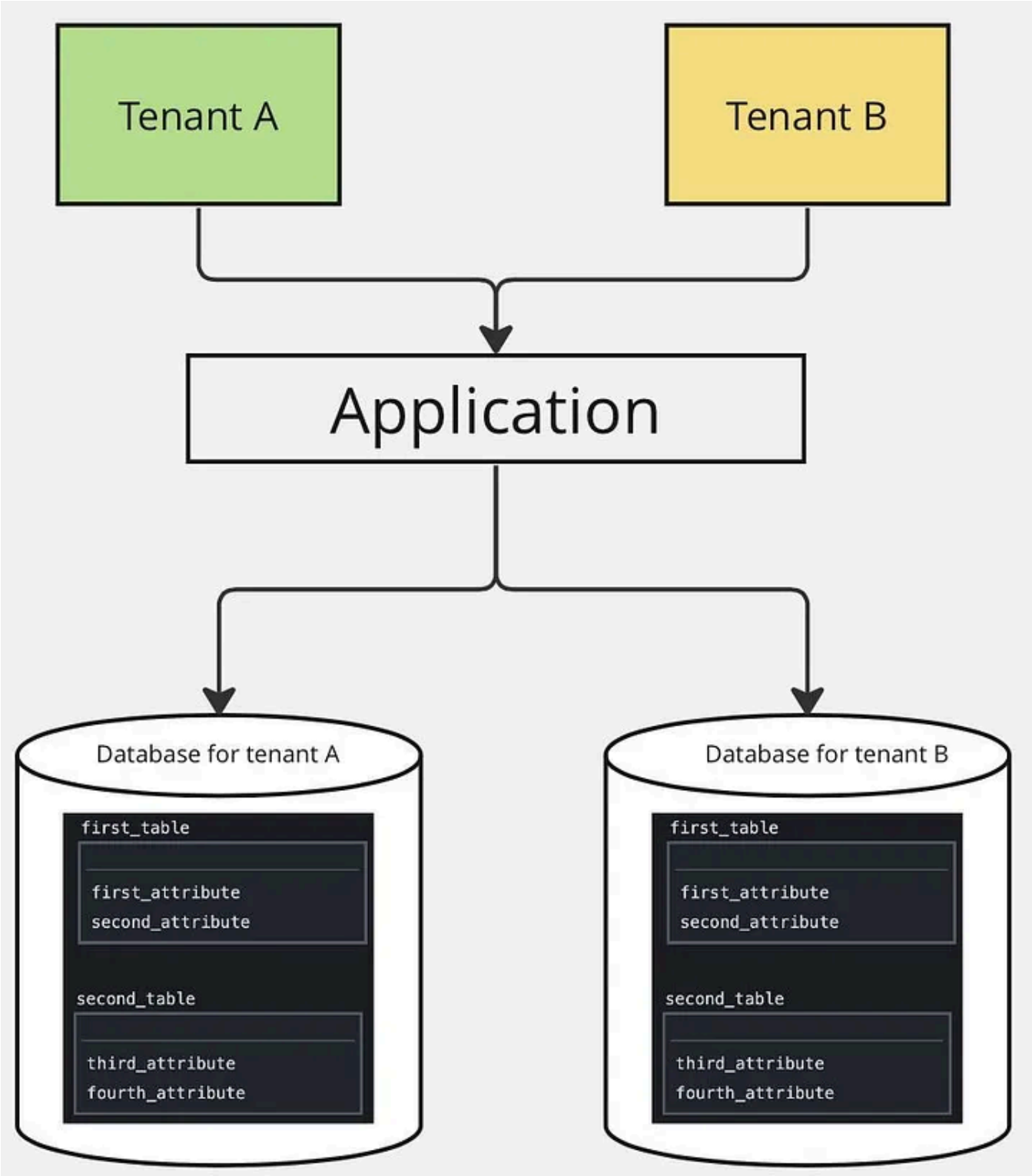
- Better database performance
- Better data separation

Disadvantages:

- More complicated in development

Database-per-tenant

This approach provides the best separation of tenant's data — we can store it in different databases or even use separate db instances:



Advantages:

- Also better database performance
- Best data separation (awesome when security requirements are strict)

Disadvantages:

- The most complicated in development
- In case of using different database instances — the highest infrastructure price

General overview of example project

Project overview

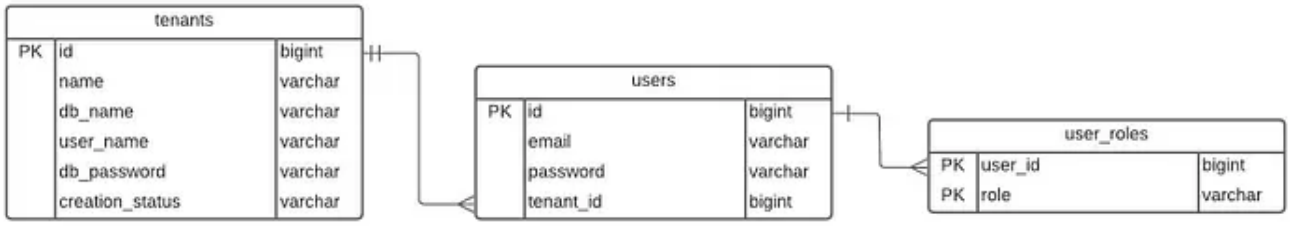
For a multi-tenancy demonstration I'm going to build a medical lab simulation, where information about research for different companies will be stored in separate databases (due to high security requirements).

Application is going to be multi-module, and 5 modules will be created:

- auth — there will be stored all the logic needed for authentication and authorization flows
- tenant-management — module for operating tenants and their databases
- lab — module for demo business logic
- commons — module for storing some parts of project which are going to be needed in few different modules
- application — main module of application, general configuration will be stored in it

Database schemas

For storing info about tenants and users I'll use main database instance with such schema:



And for storing data for each tenant databases with quite simple schema will be used:

researches		
PK	id	bigint
	name	varchar
	description	varchar

DB workflow

Dynamic database creation

For providing ability of dynamic database creation, we have to follow those steps:

1. We will need some configurations, which are going to be stored in application.yml file in *resources* folder:

```yaml
spring:
  profiles:
    active: dev
    group:
      dev:
        - dev
        - api-docs
  liquibase:
    enabled: false
  jmx:
    enabled: false
  data:
    jpa:
      repositories:
        bootstrap-mode: deferred
  main:
    allow-bean-definition-overriding: true
  datasource:
    driverClassName: org.postgresql.Driver
    url: jdbc:postgresql://127.0.0.1:5432/demo_lab
    username: demo_lab
    password: mega_secure_password
    hikari:
      minimum-idle: 90
      maximum-pool-size: 90
  jpa:
    show-sql: false
    open-in-view: false
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    properties:
      hibernate.jdbc.time_zone: UTC
      hibernate.id.new_generator_mappings: true
      hibernate.cache.use_second_level_cache: false
      hibernate.cache.use_query_cache: false
      hibernate.generate_statistics: false
      hibernate.jdbc.batch_size: 100
      hibernate.order_inserts: true
      hibernate.order_updates: true
      hibernate.query.fail_on_pagination_over_collection_fetch: true
      hibernate.query.in_clause_parameter_padding: true
    hibernate:
      ddl-auto: none
      naming:
        physical-strategy: org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNam
        implicit-strategy: org.springframework.boot.orm.jpa.hibernate.SpringImplicitNam

logging:
  level:
```

```yaml
    org.hibernate.SQL: INFO
    org.springframework.jdbc.core: INFO

datasource:
  base-url: ${DATASOURCE_URL:jdbc:postgresql://localhost:5432/}
  main:
    name: ${MAIN_DB_NAME:demo_lab}
    driver: ${MAIN_DATASOURCE_DRIVER:org.postgresql.Driver}
    url: ${MAIN_DATASOURCE_URL:jdbc:postgresql://localhost:5432/demo_lab}
    username: ${MAIN_DATASOURCE_USERNAME:demo_lab}
    password: ${MAIN_DATASOURCE_PASSWORD:mega_secure_password}

driver: org.postgresql.Driver
url: ${spring.datasource.url}
username: ${spring.datasource.username}
password: ${spring.datasource.password}
```


2. Actutally for database creation, we need to implement TenantDao:

```java
package com.konstde00.tenant_management.repository.dao;

import com.konstde00.commons.domain.enums.DatabaseCreationStatus;
import com.konstde00.tenant_management.domain.dto.data_source.TenantDbInfoDto;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;

import javax.sql.DataSource;
import java.util.List;

@Slf4j
public class TenantDao extends AbstractDao {

    @Autowired
    public TenantDao(@Qualifier("mainDataSource") DataSource mainDataSource) {
        super(mainDataSource);
    }

    public List<TenantDbInfoDto> getTenantDbInfo(DatabaseCreationStatus creationStatus)

        String query = "select id, db_name, user_name, db_password " +
                "from tenants " +
                "where creation_status = :creationStatus";

        MapSqlParameterSource params = new MapSqlParameterSource("creationStatus", crea

        return namedParameterJdbcTemplate.query(query, params, (rs, rowNum) -> {

            TenantDbInfoDto dto = new TenantDbInfoDto();

            dto.setId(rs.getLong("id"));
            dto.setDbName(rs.getString("db_name"));
            dto.setUserName(rs.getString("user_name"));
            dto.setDbPassword(rs.getString("db_password"));

            return dto;
        });
    }

    public void createTenantDb(String dbName, String userName, String password) {

        createUserIfMissing(userName, password);

        String createDbQuery = "CREATE DATABASE " + dbName;

        jdbcTemplate.execute(createDbQuery);
```

```
        log.info("Created database: " + dbName);

        String grantPrivilegesQuery = String.format("GRANT ALL PRIVILEGES ON DATABASE %

        jdbcTemplate.execute(grantPrivilegesQuery);
    }

    private void createUserIfMissing(String userName, String password) {

        try {

            String createUserQuery = String.format("""
                DO
                            $do$
                                BEGIN
                                    IF EXISTS (SELECT FROM pg_catalog.pg_roles WHERE ro
                                       ALTER USER "%s" WITH PASSWORD '%s';
                                        CREATE USER "%s" WITH CREATEDB CREATEROLE PASSW
                                    END IF;
                                END
                            $do$""", userName, userName, password, userName, password);

            jdbcTemplate.execute(createUserQuery);

        } catch (Exception exception) {

            log.error("Error during creation user : {}", exception.getMessage());
        }
    }
}
```

Dao here stands for “Data Acess Object”.

3. After db is created, we need to enable migrations. For this purpose, lets add LiquibaseService

```java
package com.konstde00.tenant_management.service;

import com.konstde00.commons.domain.entity.Tenant;
import liquibase.Contexts;
import liquibase.LabelExpression;
import liquibase.Liquibase;
import liquibase.database.Database;
import liquibase.database.DatabaseFactory;
import liquibase.database.jvm.JdbcConnection;
import liquibase.exception.DatabaseException;
import liquibase.resource.ClassLoaderResourceAccessor;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import static lombok.AccessLevel.PRIVATE;

@Slf4j
@Service
@DependsOn("dataSourceRouting")
@FieldDefaults(level = PRIVATE, makeFinal = true)
public class LiquibaseService implements SmartInitializingSingleton {

    DataSource dataSource;
    TenantService tenantService;
    ConnectionService connectionService;

    public LiquibaseService(@Qualifier("mainDataSource") DataSource dataSource,
                            @Lazy TenantService tenantService,
                            ConnectionService connectionService) {
        this.dataSource = dataSource;
        this.tenantService = tenantService;
        this.connectionService = connectionService;
    }

    static String CHANGELOG_FILE = "db.changelog-master.yml";
    static String MAIN_DS_MIGRATIONS_CLASSPATH = "classpath:liquibase/migrations/main_
    static String TENANT_MIGRATIONS_CLASSPATH = "classpath:liquibase/migrations/tenant
```

```
49      public void afterSingletonsInstantiated() {
50
51          try {
52              enableMigrationsToMainDatasource(dataSource.getConnection());
53          } catch (SQLException e) {
54              throw new RuntimeException(e);
55          }
56
57          List<Tenant> tenants = tenantService.findAll();
58
59          for (Tenant tenant : tenants) {
60
61              enableMigrationsToTenantDatasource(tenant.getDbName(), tenant.getUserName(
62          }
63      }
64
65      public void enableMigrationsToTenantDatasource(String dbName, String userName, Str
66
67          try (Connection connection = connectionService.getConnection(dbName, userName,
68
69              enableMigrationsToTenantDatasource(connection);
70
71          } catch (Exception exception) {
72
73              log.error("Exception during enabling migrations to tenant datasource: {}",
74          }
75      }
76      public static void enableMigrationsToTenantDatasource(Connection connection) {
77
78          try (Liquibase liquibase = new Liquibase(TENANT_MIGRATIONS_CLASSPATH + CHANGEL
79                      new ClassLoaderResourceAccessor(), getDatabase(connection))) {
80
81              liquibase.update(new Contexts(), new LabelExpression());
82
83          } catch (Exception exception) {
84
85              log.error("Exception during enabling migrations to tenant datasource: {}",
86          }
87      }
88
89      public void enableMigrationsToMainDatasource(String dbName, String userName, Strin
90
91          try (Connection connection = connectionService.getConnection(dbName, userName,
92
93              enableMigrationsToMainDatasource(connection);
94
95          } catch (Exception exception) {
```

```
96
97                  log.error("Exception during enabling migrations to main datasource: {}", e
98              }
99          }
100
101         public static void enableMigrationsToMainDatasource(Connection connection) {
102
103             try (Liquibase liquibase = new Liquibase(MAIN_DS_MIGRATIONS_CLASSPATH + CHANGE
104                         new ClassLoaderResourceAccessor(), getDatabase(connection))) {
105
106                 liquibase.update(new Contexts(), new LabelExpression());
107
108             } catch (Exception exception) {
109
110                 log.error("Exception during enabling migrations to main datasource: {}", e
111             }
112         }
113
114         private static Database getDatabase(Connection connection) throws DatabaseExceptio
115
116             return DatabaseFactory.getInstance()
117                     .findCorrectDatabaseImplementation(new JdbcConnection(connection));
118         }
119     }
```

Here we have specified paths for main and tenant db migrations and couple of methods for enabling them.

Example of a migration:

```yaml
databaseChangeLog:
  -  preConditions:

  -  changeSet:
      id: createTenantsTable
      author:  konstde00
      changes:
        -  createTable:
             columns:
               - column:
                   name: id
                   type: bigint
               - column:
                   name: name
                   type: varchar
               - column:
                   name: db_name
```

```
                    type: varchar
                - column:
                    name: user_name
                    type: varchar
                - column:
                    name: db_password
                    type: varchar
                - column:
                    name: creation_status
                    type: varchar
              schemaName: public
              tableName: tenants

  - changeSet:
      id: createTenantsIdSequence
      author: konstde00
      changes:
        - createSequence:
            dataType: bigint
            minValue: 2
            incrementBy: 1
            schemaName: public
            sequenceName: tenants_id_seq
```

and a simple example of a changelog file:

```
databaseChangeLog:

  - include:
      file: changelog/Tenants.yml
      relativeToChangelogFile: true
  - include:
      file: changelog/Users.yml
      relativeToChangelogFile: true
  - include:
      file: changelog/UserRoles.yml
      relativeToChangelogFile: true

#  Constraints

  - include:
      file: changelog/TenantsConstraints.yml
      relativeToChangelogFile: true
  - include:
      file: changelog/UsersConstraints.yml
      relativeToChangelogFile: true
  - include:
```

```
        file: changelog/UserRolesConstraints.yml
        relativeToChangelogFile: true
```

Also, for creating database connections we need to implement **ConnectionService**:

```java
package com.konstde00.tenant_management.service;

import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.testcontainers.containers.PostgreSQLContainer;

import java.net.ConnectException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

@Slf4j
@Component
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class ConnectionService {

    @NonFinal
    @Value("${datasource.base-url}")
    String datasourceBaseUrl;

    @NonFinal
    @Value("${datasource.main.driver}")
    String mainDatasourceDriverClassName;

    static String USER = "user";
    static String PASSWORD = "password";

    public static Connection getConnection(PostgreSQLContainer container) throws Connec

        try {

            Properties dbProperties = new Properties();
            Class.forName(container.getDriverClassName());
            dbProperties.put(USER, container.getUsername());
            dbProperties.put(PASSWORD, container.getPassword());

            return DriverManager
                .getConnection(container.getJdbcUrl(),
                    dbProperties);

        } catch (SQLException | ClassNotFoundException e) {

            log.error(e.getMessage());
```

```
49
50              throw new ConnectException("Can't connect to DB");
51          }
52      }
53
54      public Connection getConnection(String dbName, String userName, String dbPassword)
55
56          try {
57
58              Properties dbProperties = new Properties();
59
60              Class.forName(mainDatasourceDriverClassName);
61              dbProperties.put(USER, userName);
62              dbProperties.put(PASSWORD, dbPassword);
63
64              return DriverManager.getConnection(datasourceBaseUrl + dbName,
65                              dbProperties);
66
67          } catch (SQLException | ClassNotFoundException e) {
68
69              log.error(e.getMessage());
70
71              throw new ConnectException("Can't connect to DB");
72          }
73      }
74  }
```

4. For creating admin user for our tenants we have to implement **UserDao**, **UserRepository** and **UserService**:

```java
package com.konstde00.tenant_management.repository.dao;

import com.konstde00.tenant_management.domain.dto.response.UserAuthShortDto;
import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.stereotype.Repository;

import javax.sql.DataSource;
import java.util.List;

@Slf4j
@Repository
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class UserDao extends AbstractDao {

    @Autowired
    public UserDao(@Qualifier("mainDataSource") DataSource dataSource) {
        super(dataSource);
    }

    public UserAuthShortDto getAuthShortDtoByUserId(Long userId) {

        UserAuthShortDto dto = new UserAuthShortDto();

        String queryForTenantKey = """

            select u.tenant_id
            from users u
            where u.id = :userId

            """;

        MapSqlParameterSource params = new MapSqlParameterSource("userId", userId);

        Long tenantId = namedParameterJdbcTemplate.queryForObject(queryForTenantKey, pa
                -> rs.getLong("tenant_id"));

        dto.setTenantId(tenantId);
        dto.setRoles(getAuthoritiesByUserId(userId));

        return dto;
    }

    public List<String> getAuthoritiesByUserId(Long userId) {
```

```java

        String queryForAuthorities = """

            select ur.role
            from user_roles ur
            where ur.user_id = :userId

            """;

        MapSqlParameterSource params = new MapSqlParameterSource("userId", userId);

        return namedParameterJdbcTemplate.query(queryForAuthorities, params,
                (rs, rowNum) -> rs.getString("role"));
    }
}
```

UserDao.java hosted with ❤ by GitHub view raw

```java
package com.konstde00.tenant_management.repository;

import com.konstde00.commons.domain.entity.User;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {

    @EntityGraph(attributePaths = {"roles"})
    Optional<User> findByEmail(String email);
}
```

UserRepository.java hosted with ❤ by **GitHub** **view raw**

```java
package com.konstde00.tenant_management.service;

import com.konstde00.commons.domain.entity.Tenant;
import com.konstde00.commons.domain.entity.User;
import com.konstde00.commons.domain.enums.Role;
import com.konstde00.commons.exceptions.ForbiddenException;
import com.konstde00.commons.exceptions.ResourceNotFoundException;
import com.konstde00.tenant_management.domain.dto.request.CreateUserRequestDto;
import com.konstde00.tenant_management.domain.dto.response.CreateUserResponseDto;
import com.konstde00.tenant_management.domain.dto.response.UserAuthShortDto;
import com.konstde00.tenant_management.mapper.UserMapper;
import com.konstde00.tenant_management.repository.UserRepository;
import com.konstde00.tenant_management.repository.dao.UserDao;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jws;
import io.jsonwebtoken.Jwts;
import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.StringUtils;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Lazy;
import org.springframework.security.crypto.bcrypt.BCrypt;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.servlet.http.HttpServletRequest;
import java.util.List;

import static java.lang.String.format;
import static org.springframework.http.HttpHeaders.AUTHORIZATION;

@Slf4j
@Service
@DependsOn("dataSourceRouting")
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class UserService {

    @NonFinal
    @Value("${jwt.secret}")
    String jwtSecret;

    UserDao userDao;
    TenantService tenantService;
    UserRepository userRepository;

```

```
49      public UserService(UserDao userDao,
50                         @Lazy TenantService tenantService,
51                         UserRepository userRepository) {
52          this.userDao = userDao;
53          this.tenantService = tenantService;
54          this.userRepository = userRepository;
55      }
56
57      public User getByEmail(String email) {
58
59          return userRepository.findByEmail(email)
60                  .orElseThrow(() -> {
61                      log.error(format("User with email - %s does not exist. ", email));
62                      return new ResourceNotFoundException(format("User with email - %s
63                  });
64      }
65
66      public CreateUserResponseDto create(CreateUserRequestDto requestDto) {
67
68          Tenant tenant = tenantService.getById(requestDto.getTenantId());
69
70          return create(requestDto, tenant, requestDto.getRoles());
71      }
72
73      public CreateUserResponseDto create(CreateUserRequestDto requestDto, Tenant tenant
74
75          User user = UserMapper.INSTANCE.fromRequestDto(requestDto);
76
77          user.setRoles(roles);
78          user.setTenant(tenant);
79          user.setPassword(bcryptPassword(requestDto.getPassword()));
80
81          user = userRepository.saveAndFlush(user);
82
83          return UserMapper.INSTANCE.toResponseDto(user);
84      }
85
86      private String bcryptPassword(String password) {
87          return BCrypt.hashpw(password, BCrypt.gensalt());
88      }
89
90      @Transactional
91      public UserAuthShortDto getActualUser(HttpServletRequest request) {
92          String token = request.getHeader(AUTHORIZATION);
93
94          if (token != null && token.startsWith("Bearer ")) {
95
```

```
96              try {
97                  String claims = token.replace("Bearer ", StringUtils.EMPTY);
98
99                  Jws<Claims> claimsJws = Jwts.parserBuilder()
100                         .setSigningKey(jwtSecret.getBytes())
101                         .build()
102                         .parseClaimsJws(claims);
103
104                 Long userId = Long.parseLong(claimsJws.getBody().getSubject());
105                 UserAuthShortDto user = userDao.getAuthShortDtoByUserId(userId);
106
107                 if (user.getRoles() == null || user.getRoles().isEmpty()) {
108
109                     throw new ForbiddenException("Access denied");
110                 }
111
112                 return user;
113
114             } catch (Exception e) {
115
116                 log.error("Exception occurred while 'getActualUser' execution: " + e.g
117
118                 throw new ForbiddenException("Access denied");
119             }
120         }
121         throw new ForbiddenException("Access denied");
122     }
123 }
```

5. For operating migration of Main and Tenant datasources, lets add **DataSourceConfigService**:

```java
package com.konstde00.tenant_management.service.data_source;

import com.konstde00.tenant_management.domain.dto.data_source.TenantDbInfoDto;
import com.konstde00.tenant_management.repository.dao.TenantDao;
import com.konstde00.tenant_management.service.LiquibaseService;
import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static com.konstde00.commons.domain.enums.DatabaseCreationStatus.CREATED;

@Slf4j
@Service
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class DataSourceConfigService {

    @NonFinal
    @Value("${datasource.main.name}")
    String mainDatasourceName;

    @NonFinal
    @Value("${datasource.main.username}")
    String mainDatasourceUsername;

    @NonFinal
    @Value("${datasource.main.password}")
    String mainDatasourcePassword;

    @NonFinal
    @Value("${datasource.base-url}")
    String datasourceBaseUrl;

    @NonFinal
    Boolean wasMainDatasourceConfigured = false;

    DataSource mainDataSource;
    LiquibaseService liquibaseService;

```

```
    public DataSourceConfigService(@Qualifier("mainDataSource") DataSource mainDataSour
                                   LiquibaseService liquibaseService) {
        this.mainDataSource = mainDataSource;
        this.liquibaseService = liquibaseService;
    }

    public Map<Object, Object> configureDataSources() {

        Map<Object, Object> dataSources = new HashMap<>();

        if (!wasMainDatasourceConfigured)  {
            liquibaseService.enableMigrationsToMainDatasource(mainDatasourceName,
                    mainDatasourceUsername, mainDatasourcePassword);
            wasMainDatasourceConfigured = true;
        }

        List<TenantDbInfoDto> dtos = new TenantDao(mainDataSource).getTenantDbInfo(CREA

        dataSources.put(null, mainDataSource);
        for (TenantDbInfoDto dto : dtos) {

            dataSources.put(dto.getId(), configureDataSource(dto));
        }

        return dataSources;
    }

    private DataSource configureDataSource(TenantDbInfoDto dto) {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setUrl(getUrl(dto));
        dataSource.setUsername(dto.getUserName());
        dataSource.setPassword(dto.getDbPassword());

        return dataSource;
    }

    private String getUrl(TenantDbInfoDto dto) {

        return datasourceBaseUrl + dto.getDbName();
    }
}
```

6. To provide an opportunity of choosing different datasources we have to add **DataSourceRoutingService:**

```java
package com.konstde00.tenant_management.service.data_source;

import com.konstde00.tenant_management.service.LiquibaseService;
import com.konstde00.tenant_management.service.dao_holder.AbstractDaoHolder;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.context.annotation.Lazy;
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.Map;
import static lombok.AccessLevel.PRIVATE;

@Slf4j
@Service(value = "dataSourceRouting")
@FieldDefaults(level = PRIVATE, makeFinal = true)
public class DataSourceRoutingService extends AbstractRoutingDataSource implements Smar

    LiquibaseService liquibaseService;
    Map<String, AbstractDaoHolder> daoHolders;
    DataSourceConfigService datasourceConfigService;

    @NonFinal
    @Value("${datasource.main.name}")
    String mainDatasourceName;

    @NonFinal
    @Value("${datasource.main.username}")
    String mainDatasourceUsername;

    @NonFinal
    @Value("${datasource.main.password}")
    String mainDatasourcePassword;

    public DataSourceRoutingService(@Lazy DataSourceConfigService datasourceConfigServi
                                    LiquibaseService liquibaseService,
                                    @Qualifier("mainDataSource") DataSource mainDataSou
                                    Map<String, AbstractDaoHolder> daoHolders) {
        this.datasourceConfigService = datasourceConfigService;

        this.liquibaseService = liquibaseService;
        this.liquibaseService.enableMigrationsToMainDatasource(mainDatasourceName,
```

```
                mainDatasourceUsername, mainDatasourcePassword);

        Map<Object, Object> dataSourceMap = this.datasourceConfigService.configureDataS

        this.setTargetDataSources(dataSourceMap);
        this.setDefaultTargetDataSource(mainDataSource);

        this.daoHolders = daoHolders;
    }

    @Override
    public void afterSingletonsInstantiated() {

        Map<Object, Object> dataSources
                = datasourceConfigService.configureDataSources();

        updateResolvedDataSources(dataSources);

        updateDaoTemplateHolders(dataSources);
    }

    @Override
    protected Long determineCurrentLookupKey() {

        return DataSourceContextHolder.getCurrentTenantId();
    }

    public void updateResolvedDataSources(Map<Object, Object> dataSources) {

        setTargetDataSources(dataSources);

        afterPropertiesSet();
    }

    public void updateDaoTemplateHolders(Map<Object, Object> dataSources) {

        daoHolders.forEach((key, value) -> value.addNewTemplates(dataSources));
    }
}
```

For this service important to explain the concept of **target** and **resolved** data sources:

In **AbstractRoutingDataSource** exist 2 important properties:

```
@Nullable
private Map<Object, Object> targetDataSources;
```

and

```
@Nullable
private Map<Object, DataSource> resolvedDataSources;
```

Without reflection-api we can set only target datasources (using *setTargetDataSources* method), but when we switch between different data sources we use **resolvedDataSources** map. To update this property, we can use *afterPropertiesSet* method from **AbstractRoutingDataSource**:

```
@Override
public void afterPropertiesSet() {
   if (this.targetDataSources == null) {
      throw new IllegalArgumentException("Property 'targetDataSources' is requi
   }
   this.resolvedDataSources = CollectionUtils.newHashMap(this.targetDataSources
   this.targetDataSources.forEach((key, value) -> {
      Object lookupKey = resolveSpecifiedLookupKey(key);
      DataSource dataSource = resolveSpecifiedDataSource(value);
      this.resolvedDataSources.put(lookupKey, dataSource);
   });
   if (this.defaultTargetDataSource != null) {
      this.resolvedDefaultDataSource = resolveSpecifiedDataSource(this.defaultT
   }
}
```

in this method we iterate through target datasources and update resolved datasources using them. So in our **DataSourceRoutingService** to update list of available datasources we need only to call this 2 methods like this:

```
public void updateResolvedDataSources(Map<Object, Object> dataSources) {
```

```
        setTargetDataSources(dataSources);

        afterPropertiesSet();
    }
```

7. Then lets add **TenantMapper:**

```
package com.konstde00.tenant_management.mapper;

import com.konstde00.commons.domain.entity.Tenant;
import com.konstde00.tenant_management.domain.dto.request.CreateTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.response.TenantResponseDto;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;

import java.util.List;

@Mapper
public interface TenantMapper {

    TenantMapper INSTANCE = Mappers.getMapper(TenantMapper.class);

    Tenant fromRequestDto(CreateTenantRequestDto requestDto);

    @Mapping(target = "userId", ignore = true)
    TenantResponseDto toResponseDto(Tenant tenant);

    List<TenantResponseDto> toResponseDtoList(List<Tenant> tenants);
}
```

TenantMapper.java hosted with ❤ by **GitHub** **view raw**

8. Next, we have to add **TenantService.** It will be used for performing the majority operations with “Tenant” entity:

```java
package com.konstde00.tenant_management.service;

import com.konstde00.commons.domain.entity.Tenant;
import com.konstde00.commons.domain.enums.Role;
import com.konstde00.commons.exceptions.NotValidException;
import com.konstde00.tenant_management.domain.dto.request.CreateTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.request.RenameTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.response.TenantResponseDto;
import com.konstde00.tenant_management.mapper.TenantMapper;
import com.konstde00.tenant_management.repository.TenantRepository;
import com.konstde00.tenant_management.repository.dao.TenantDao;
import com.konstde00.tenant_management.service.data_source.DataSourceRoutingService;
import com.konstde00.tenant_management.service.data_source.DataSourceConfigService;
import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.DependsOn;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.sql.DataSource;
import java.util.List;
import java.util.Map;

import static com.konstde00.commons.domain.enums.DatabaseCreationStatus.*;

@Slf4j
@Service
@DependsOn("dataSourceRouting")
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class TenantService {

    TenantDao tenantDao;
    UserService userService;
    LiquibaseService liquibaseService;
    TenantRepository tenantRepository;
    DataSourceConfigService datasourceConfigService;
    DataSourceRoutingService dataSourceRoutingService;

    public TenantService(UserService userService,
                         TenantRepository tenantRepository,
                         LiquibaseService liquibaseService,
                         @Qualifier("mainDataSource") DataSource mainDatasource,
                         DataSourceConfigService datasourceConfigService,
                         DataSourceRoutingService dataSourceRoutingService) {
        this.userService = userService;
        this.tenantRepository = tenantRepository;
```

```
        this.liquibaseService = liquibaseService;
        this.tenantDao = new TenantDao(mainDatasource);
        this.datasourceConfigService = datasourceConfigService;
        this.dataSourceRoutingService = dataSourceRoutingService;
    }

    public Tenant getById(Long id) {

        return tenantRepository.findById(id)
                .orElseThrow(() -> new NotValidException("Can't find tenant by id " +
    }

    public List<Tenant> findAll() {

        return tenantRepository.findAll();
    }

    public TenantResponseDto create(CreateTenantRequestDto requestDto) {

        Tenant tenant = TenantMapper.INSTANCE.fromRequestDto(requestDto);

        tenant.setCreationStatus(IN_PROGRESS);
        tenant = saveAndFlush(tenant);

        try {

            tenantDao.createTenantDb(requestDto.getName(), requestDto.getUserName(), r
            tenant.setCreationStatus(CREATED);

        } catch (Exception e) {

            log.error("Failed to create tenant db: " + e.getMessage());
            tenant.setCreationStatus(FAILED_TO_CREATE);

        } finally {

            tenant = saveAndFlush(tenant);
        }

        TenantResponseDto responseDto = TenantMapper.INSTANCE.toResponseDto(tenant);

        if (CREATED.equals(tenant.getCreationStatus())) {

            liquibaseService.enableMigrationsToTenantDatasource(requestDto.getDbName()

            Long userId = userService.create(requestDto.getUser(), tenant, List.of(Rol
            responseDto.setUserId(userId);
```

```

            Map<Object, Object> configuredDataSources = datasourceConfigService
                    .configureDataSources();

            dataSourceRoutingService.updateResolvedDataSources(configuredDataSources);
        }

        return responseDto;
    }

    public Tenant saveAndFlush(Tenant tenant) {

        return tenantRepository.saveAndFlush(tenant);
    }

    @Transactional
    public void rename(RenameTenantRequestDto params) {

        tenantRepository.rename(params.getId(), params.getName());
    }

    @Transactional
    public void delete(Long id) {

        tenantRepository.deleteById(id);
    }
}
```

Dynamic database choosing

- Spring Data JPA

Firstly lets add the following dependencies to the *pom.xml* file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

spring-boot-data-jpa hosted with ❤ by **GitHub** **view raw**

In my example app I use PostgreSql, this dependency needs to be added:

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

postgresql hosted with ❤ by **GitHub** **view raw**

Implement **DataSourceContextHolder:**

```java
package com.konstde00.tenant_management.service.data_source;

import com.konstde00.tenant_management.domain.dto.response.UserAuthShortDto;
import com.konstde00.tenant_management.service.UserService;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;

import static lombok.AccessLevel.PRIVATE;

@Slf4j
@Component
@FieldDefaults(level = PRIVATE, makeFinal = true)
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public class DataSourceContextHolder {

    UserService userService;

    @NonFinal
    static ThreadLocal<Long> currentTenantId = new ThreadLocal<>();

    static Long DEFAULT_TENANT_ID = null;

    public DataSourceContextHolder(@Lazy UserService userService) {

        this.userService = userService;
    }

    public static void setCurrentTenantId(Long tenantId) {

        currentTenantId.set(tenantId);
    }

    public static Long getCurrentTenantId() {

        return currentTenantId.get();
    }

    public void updateTenantContext(HttpServletRequest request) {

        Long tenantId;

```

```
        try {

            UserAuthShortDto user = userService.getActualUser(request);

            tenantId = user.getTenantId();

            setCurrentTenantId(tenantId);

        } catch (Exception e) {

            log.error("Exception occurred while 'updateTenantContext' execution: " + e.

           tenantId = DEFAULT_TENANT_ID;
        }

        setCurrentTenantId(tenantId);
    }
}
```

Then we need to add **DataSourceRoutingService:**

```java
package com.konstde00.tenant_management.service.data_source;

import com.konstde00.tenant_management.service.LiquibaseService;
import com.konstde00.tenant_management.service.dao_holder.AbstractDaoHolder;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.context.annotation.Lazy;
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.Map;
import static lombok.AccessLevel.PRIVATE;

@Slf4j
@RefreshScope
@Service(value = "dataSourceRouting")
@FieldDefaults(level = PRIVATE, makeFinal = true)
public class DataSourceRoutingService extends AbstractRoutingDataSource implements Smar

    LiquibaseService liquibaseService;
    Map<String, AbstractDaoHolder> daoHolders;
    DataSourceConfigService datasourceConfigService;

    @NonFinal
    @Value("${datasource.main.name}")
    String mainDatasourceName;

    @NonFinal
    @Value("${datasource.main.username}")
    String mainDatasourceUsername;

    @NonFinal
    @Value("${datasource.main.password}")
    String mainDatasourcePassword;

    public DataSourceRoutingService(@Lazy DataSourceConfigService datasourceConfigServi
                                    LiquibaseService liquibaseService,
                                    @Qualifier("mainDataSource") DataSource mainDataSou
                                    Map<String, AbstractDaoHolder> daoHolders) {
        this.datasourceConfigService = datasourceConfigService;

        this.liquibaseService = liquibaseService;
```

```
        this.liquibaseService.enableMigrationsToMainDatasource(mainDatasourceName,
                mainDatasourceUsername, mainDatasourcePassword);

        Map<Object, Object> dataSourceMap = this.datasourceConfigService.configureDataS

        this.setTargetDataSources(dataSourceMap);
        this.setDefaultTargetDataSource(mainDataSource);

        this.daoHolders = daoHolders;
    }

    @Override
    public void afterSingletonsInstantiated() {

        Map<Object, Object> dataSources
                = datasourceConfigService.configureDataSources();

        updateResolvedDataSources(dataSources);

        updateDaoTemplateHolders(dataSources);
    }

    @Override
    protected Long determineCurrentLookupKey() {

        return DataSourceContextHolder.getCurrentTenantId();
    }

    public void updateResolvedDataSources(Map<Object, Object> dataSources) {

        setTargetDataSources(dataSources);

        afterPropertiesSet();
    }

    public void updateDaoTemplateHolders(Map<Object, Object> dataSources) {

        daoHolders.forEach((key, value) -> value.addNewTemplates(dataSources));
    }
}
```

Next, create **DataSourceConfig:**

```java
package com.konstde00.tenant_management.service.data_source;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Primary;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@EnableTransactionManagement
@DependsOn("dataSourceRouting")
public class DataSourceConfig {

    private DataSourceRoutingService dataSourceRouting;

    public DataSourceConfig(DataSourceRoutingService dataSourceRouting) {
        this.dataSourceRouting = dataSourceRouting;
    }

    @Bean
    @Primary
    public DataSource dataSource() {
        return dataSourceRouting;
    }

    @Primary
    @Bean(name="customEntityManager")
    public LocalContainerEntityManagerFactoryBean entityManagerBean(EntityManagerFactor
        return builder.dataSource(dataSource()).packages("com.konstde00.auth", "com.kon
                "com.konstde00.tenant_management", "com.konstde00.lab", "com.konstde00.
    }

    @Bean(name="customEntityManagerFactory")
    public LocalContainerEntityManagerFactoryBean customEntityManagerFactoryBean(Entity
        return builder.dataSource(dataSource()).packages("com.konstde00.auth", "com.kon
                "com.konstde00.tenant_management", "com.konstde00.lab", "com.konstde00.
    }

    @Bean(name = "customTransactionManager")
    public JpaTransactionManager transactionManager(
        @Autowired @Qualifier("customEntityManager") LocalContainerEntityManagerFactory
```

```
        return new JpaTransactionManager(customEntityManagerFactoryBean.getObject());
    }
}
```

DataSourceConfig.java hosted with ❤ by GitHub view raw

Then, we need to implement **TenantsRoutingFilter**

```
package com.konstde00.tenant_management.config;

import com.konstde00.tenant_management.service.data_source.DataSourceContextHolder;
import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Slf4j
@Order(2) // the order has to be updated in case of using different amount of filters
@Component
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class TenantsRoutingFilter extends OncePerRequestFilter {

    DataSourceContextHolder dataSourceContextHolder;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse res

        dataSourceContextHolder.updateTenantContext(request);

        filterChain.doFilter(request, response);
    }
}
```

TenantsRoutingFilter.java hosted with ❤ by **GitHub** **view raw**

Finally, lets add **TenantController** with basic *CRUD* operations with tenants:

```java
package com.konstde00.tenant_management.controller;

import com.konstde00.commons.domain.entity.Tenant;
import com.konstde00.tenant_management.domain.dto.request.CreateTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.request.RenameTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.response.TenantResponseDto;
import com.konstde00.tenant_management.mapper.TenantMapper;
import com.konstde00.tenant_management.service.TenantService;
import io.swagger.v3.oas.annotations.Operation;
import lombok.AllArgsConstructor;
import lombok.NonNull;
import lombok.experimental.FieldDefaults;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

import static lombok.AccessLevel.PRIVATE;

@RestController
@AllArgsConstructor
@RequestMapping("/api/tenants")
@FieldDefaults(level = PRIVATE, makeFinal = true)
public class TenantController {

    TenantService tenantService;

    @GetMapping("/v1")
    @Operation(summary = "Get all tenants")
    public ResponseEntity<List<TenantResponseDto>> getAllTenants() {

        List<Tenant> tenants = tenantService.findAll();

        List<TenantResponseDto> responseDtos
            = TenantMapper.INSTANCE.toResponseDtoList(tenants);

        return new ResponseEntity<>(responseDtos, HttpStatus.OK);
    }

    @PostMapping("/v1")
    @Operation(summary = "Create a new tenant")
    public ResponseEntity<TenantResponseDto> createTenant(@RequestBody CreateTenantRequ

        TenantResponseDto tenant = tenantService.create(tenantDto);

        return new ResponseEntity<>(tenant, HttpStatus.CREATED);
    }
```

```
    }

    @PatchMapping("/v1")
    @Operation(summary = "Rename tenant")
    public ResponseEntity<?> renameTenant(@RequestBody RenameTenantRequestDto params) {

        tenantService.rename(params);

        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }

    @DeleteMapping("/v1")
    @Operation(summary = "Delete tenant")
    public ResponseEntity<?> deleteTenant(@RequestParam @NonNull Long id) {

        tenantService.delete(id);

        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

- **Spring Data JDBC**

The idea of using Spring Data JDBC here is pretty simple and comes from Strategy pattern: we will operate multiple Dao-classes, each of them will be needed to access data in datasource of a particular tenant. We will also implement Dao-Holders, which will be used for operating those Dao-classes, and use them every time we will need an access to DB of particular tenant.

Firstly, let's add following dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

spring-boot-started-data-jdbc hosted with ❤ by **GitHub** **view raw**

Then implement AbstactDao, the parent for all other Dao classes:

```java
package com.konstde00.tenant_management.repository.dao;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import javax.sql.DataSource;

@Slf4j
@Data
@Repository
@AllArgsConstructor
@FieldDefaults(level = AccessLevel.PROTECTED, makeFinal = true)
public abstract class AbstractDao {

    JdbcTemplate jdbcTemplate;
    NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    @Autowired
    protected AbstractDao(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
    }
}
```

AbstractDao.java hosted with ❤ by **GitHub** **view raw**

Then create ResearchDao, the ‘concrete Dao’ class for operating Researches:

```java
package com.konstde00.lab.repository.dao;

import com.konstde00.lab.domain.dto.request.ResearchDto;
import com.konstde00.tenant_management.repository.dao.AbstractDao;

import javax.sql.DataSource;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

public class ResearchDao extends AbstractDao {

    protected ResearchDao(DataSource dataSource) {
        super(dataSource);
    }

    public List<ResearchDto> findAll() {

        String query = """

                select id, name, description
                from researches

                """;

        return namedParameterJdbcTemplate.query(query, (rs, rowNum) -> toDto(rs));
    }

    private ResearchDto toDto(ResultSet resultSet) throws SQLException {

        return ResearchDto
                .builder()
                .id(resultSet.getLong("id"))
                .name(resultSet.getString("name"))
                .description(resultSet.getString("description"))
                .build();
    }
}
```

ResearchDao.java hosted with ❤ by **GitHub** **view raw**

After that we will need to create **AbstractDaoHolder** and **ResearchDaoHolder**, where methods for operating **ResearchDao** instances will be created:

```java
package com.konstde00.tenant_management.service.dao_holder;

import com.konstde00.tenant_management.repository.dao.TenantDao;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.Map;

import static lombok.AccessLevel.PROTECTED;

@Service
@FieldDefaults(level = PROTECTED, makeFinal = true)
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public abstract class AbstractDaoHolder implements SmartInitializingSingleton {

    @NonFinal
    Map<Long, TenantDao> templates;

    public abstract void addNewTemplates(Map<Object, Object> dataSources);
}
```

```java
package com.konstde00.lab.service.dao_holder;

import com.konstde00.tenant_management.repository.dao.TenantDao;
import com.konstde00.tenant_management.service.dao_holder.AbstractDaoHolder;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class ResearchDaoHolder extends AbstractDaoHolder {

    @Override
    public void afterSingletonsInstantiated() {

        templates = new HashMap<>();
    }

    public TenantDao getTemplateByTenantKey(Long tenantKey) {

        return templates.get(tenantKey);
    }

    public void addNewTemplates(Map<Object, Object> dataSources) {

        dataSources.forEach((key, value) -> {

            TenantDao tenantDao = new TenantDao((DataSource) value);

            templates.putIfAbsent((Long) key, tenantDao);
        });
    }
}
```

ResearchDaoHolder.java hosted with ❤ by **GitHub** **view raw**

Then we have to make some DataSourceRoutingService: add a

```
Map<String, AbstractDaoHolder> daoHolders
```

field and a method

```
public void updateDaoHolders(Map<Object, DataSource> dataSources) {

    daoHolders.forEach((key, value) -> value.addNewTemplates(dataSources));
}
```

to update Dao holders, so now it looks like this:

```java
package com.konstde00.tenant_management.service.data_source;

import com.konstde00.tenant_management.service.LiquibaseService;
import com.konstde00.tenant_management.service.dao_holder.AbstractDaoHolder;
import lombok.experimental.FieldDefaults;
import lombok.experimental.NonFinal;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.context.annotation.Lazy;
import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
import org.springframework.stereotype.Service;

import javax.sql.DataSource;
import java.util.Map;
import static lombok.AccessLevel.PRIVATE;

@Slf4j
@RefreshScope
@Service(value = "dataSourceRouting")
@FieldDefaults(level = PRIVATE, makeFinal = true)
public class DataSourceRoutingService extends AbstractRoutingDataSource implements Smar

    LiquibaseService liquibaseService;
    Map<String, AbstractDaoHolder> daoHolders;
    DataSourceConfigService datasourceConfigService;

    @NonFinal
    @Value("${datasource.main.name}")
    String mainDatasourceName;

    @NonFinal
    @Value("${datasource.main.username}")
    String mainDatasourceUsername;

    @NonFinal
    @Value("${datasource.main.password}")
    String mainDatasourcePassword;

    public DataSourceRoutingService(@Lazy DataSourceConfigService datasourceConfigServi
                                    LiquibaseService liquibaseService,
                                    @Qualifier("mainDataSource") DataSource mainDataSou
                                    Map<String, AbstractDaoHolder> daoHolders) {
        this.datasourceConfigService = datasourceConfigService;

        this.liquibaseService = liquibaseService;
```

```
        this.liquibaseService.enableMigrationsToMainDatasource(mainDatasourceName,
                mainDatasourceUsername, mainDatasourcePassword);

        Map<Object, Object> dataSourceMap = this.datasourceConfigService.configureDataS

        this.setTargetDataSources(dataSourceMap);
        this.setDefaultTargetDataSource(mainDataSource);

        this.daoHolders = daoHolders;
    }

    @Override
    public void afterSingletonsInstantiated() {

        Map<Object, Object> dataSources
                = datasourceConfigService.configureDataSources();

        updateResolvedDataSources(dataSources);

        updateDaoTemplateHolders(dataSources);
    }

    @Override
    protected Long determineCurrentLookupKey() {

        return DataSourceContextHolder.getCurrentTenantId();
    }

    public void updateResolvedDataSources(Map<Object, Object> dataSources) {

        setTargetDataSources(dataSources);

        afterPropertiesSet();
    }

    public void updateDaoTemplateHolders(Map<Object, Object> dataSources) {

        daoHolders.forEach((key, value) -> value.addNewTemplates(dataSources));
    }
}
```

In this way we'll have an ability to use a Dao class for new datasource as soon as it's created

Integration tests

In this section I will show how to write convenient integration tests for persistence layer using Spring Boot, Testcontainers, DbRider, Datasource Proxy.

Firstly, lets add required dependencies to our *pom.xml* file:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>testcontainers-bom</artifactId>
            <version>1.15.1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>1.17.6</version>
</dependency>
```

```
<dependency>
    <groupId>com.github.database-rider</groupId>
    <artifactId>rider-core</artifactId>
    <version>1.35.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>com.github.database-rider</groupId>
    <artifactId>rider-spring</artifactId>
    <version>1.35.0</version>
</dependency>
```

```
<dependency>
    <groupId>com.github.gavlyukovskiy</groupId>
    <artifactId>p6spy-spring-boot-starter</artifactId>
    <version>1.6.2</version>
    <scope>test</scope>
</dependency>
```

First of all, we will create **AbstractApiTest** class and hide most of configurations there. So our tests will look as follows:

```java
package com.konstde00.lab.controller;

import com.github.database.rider.core.api.dataset.DataSet;
import com.github.database.rider.core.api.dataset.ExpectedDataSet;
import com.github.database.rider.spring.api.DBRider;
import org.junit.Test;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@DBRider(dataSourceBeanName = "tenantDataSource")
public class ResearchControllerTest extends AbstractApiTest {

    @Test
    @DataSet(value = {"datasets/get_all_researches/setup.yaml"})
    @ExpectedDataSet(value = {"datasets/get_all_researches/expected.yaml"})
    public void getAllResearchesTest() throws Exception {

        String token = tokenService.generate(USER).getToken();

        mockMvc.perform(
                get("/api/researches/v1")
                        .contentType(APPLICATION_JSON)
                        .header("Authorization", "Bearer " + token))
                .andExpect(status().isOk())
                .andExpect(
                        content().json(jsonReader.read("getAllResearchesResponse.json")
                .andDo(print());
    }

    @Test
    @DataSet(value = {"datasets/create_research/setup.yaml"})
    @ExpectedDataSet(
            value = {"datasets/create_research/expected.yaml"},
            ignoreCols = {"id"}
    )
    public void createResearchTest() throws Exception {

        String token = tokenService.generate(ADMIN).getToken();

        mockMvc.perform(
                        post("/api/researches/v1")
                                .contentType(APPLICATION_JSON)
                                .header("Authorization", "Bearer " + token)
```

```
                                .content(jsonReader.read("createResearchRequest.json"))
                .andExpect(status().isCreated())
                .andDo(print());
    }
}
```

ResearchControllerTest.java hosted with ❤ by GitHub view raw

There is a great tool **Database Rider** for managing datasets. It uses another library **DbUnit** as the main engine and makes the configuration extremely easy. There are a lot of options for configuration depending on your environment, but in the case of JUnit5 and Spring Boot all you need to do is to place **@DBRider** annotation for your test and that is it (*for more information see the official documentation — luckily the documentation is pretty good*).

After that, you can place @DataSet annotation on your class/test method and use the DbUnit dataset in a preferable format (YAML, XML, JSON, CSV, XLS formats or even your own Java class). In our case, it will be YAML:

```yaml
researches:
  - id: '0'
    name: First research name
    description: First research description
```

Moreover, there is an ability to specify data which is expected to be in DB after some actions performed in the test. This could be done with @ExpectedDataSet annotation.

If we don't need to check some columns (for example auto-generated ids or some timestamps) we could specify them in ignoreCols property like this:

```
@ExpectedDataSet(
        value = {"datasets/create_research/expected.yaml"},
        ignoreCols = {"id"}
)
```

Next, let's see the content of the base abstract class:

```java
package com.konstde00.lab.controller;

import com.konstde00.application.Application;
import com.konstde00.auth.service.TokenService;
import com.konstde00.commons.domain.entity.User;
import com.konstde00.commons.domain.enums.Role;
import com.konstde00.lab.config.TestTenantConfig;
import com.konstde00.lab.util.DatabaseContainerInitializer;
import com.konstde00.lab.util.JsonReader;
import lombok.AccessLevel;
import lombok.NoArgsConstructor;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.TestInstance;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import static org.junit.jupiter.api.TestInstance.Lifecycle.PER_CLASS;
import static org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabas

@Slf4j
@NoArgsConstructor
@AutoConfigureMockMvc
@TestInstance(PER_CLASS)
@RunWith(SpringRunner.class)
@AutoConfigureTestDatabase(replace = NONE)
@FieldDefaults(level = AccessLevel.PROTECTED)
@Transactional(propagation = Propagation.NOT_SUPPORTED)
@ContextConfiguration(
        initializers = DatabaseContainerInitializer.class,
        classes = Application.class
)
@SpringBootTest(
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT
)
public class AbstractApiTest {

    @Autowired
```

```
    MockMvc mockMvc;
    @Autowired
    JsonReader jsonReader;
    @Autowired
    TokenService tokenService;

    public static final User USER = User
            .builder()
            .id(TestTenantConfig.userId)
            .roles(List.of(Role.USER))
            .build();

    public static final User ADMIN = User
            .builder()
            .id(TestTenantConfig.adminId)
            .roles(List.of(Role.ADMIN))
            .build();
}
```

Now, let's think about where we will get the database for testing. The best practice is to keep the test environment as similar to the production environment as possible. So we will use the same database as in production (PostgreSQL in our case).

Likely we have **Docker** that can bring us almost any external dependency for testing. We will go further and use **Testcontainers** library that facilitates running docker containers directly from our tests.

Also, Testcontainers provides nice wrappers for many popular products (including PostgreSQL, MySQL and some other databases). Now we can create **DatabaseContainerInitializer** as a custom Spring initializer:

```java
package com.konstde00.lab.util;

import com.github.dockerjava.api.model.RestartPolicy;
import lombok.AccessLevel;
import lombok.experimental.FieldDefaults;
import org.springframework.boot.test.util.TestPropertyValues;
import org.springframework.context.ApplicationContextInitializer;
import org.springframework.context.ConfigurableApplicationContext;
import org.testcontainers.containers.PostgreSQLContainer;

@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class DatabaseContainerInitializer implements ApplicationContextInitializer<Conf
    static String TESTCONTAINERS_DISABLE_PROPERTY = "testcontainers.disable";

    public static PostgreSQLContainer postgresMainContainer
            = new PostgreSQLContainer<>("postgres:14.1-alpine")
            .withUsername("demo_lab")
            .withPassword("mega_secure_password")
            .withDatabaseName("demo_lab")
            .withReuse(true)
            .withCreateContainerCmdModifier(cmd -> cmd.getHostConfig().withRestartPolic
            .withLabel("group", "demo_lab");

    @Override
    public void initialize(ConfigurableApplicationContext configurableApplicationContex
        if (isTestcontainersDisabled(configurableApplicationContext)) {
            return;
        }

        postgresMainContainer.start();

        TestPropertyValues.of(
                "spring.datasource.url=" + postgresMainContainer.getJdbcUrl(),
                "spring.datasource.password=" + postgresMainContainer.getPassword(),
                "spring.datasource.username=" + postgresMainContainer.getUsername(),
                "datasource.main.name=" + postgresMainContainer.getDatabaseName(),
                "datasource.main.password=" + postgresMainContainer.getPassword(),
                "datasource.main.url=" + postgresMainContainer.getJdbcUrl(),
                "datasource.base-url=" + postgresMainContainer.getJdbcUrl()
                        .replace(postgresMainContainer.getDatabaseName(), "")
        ).applyTo(configurableApplicationContext.getEnvironment());

        LiquibaseUtil.enableMigrationsToMainDatasource(DatabaseContainerInitializer.pos
    }

    private boolean isTestcontainersDisabled(ConfigurableApplicationContext configurabl
        Boolean testcontainersDisabled = configurableApplicationContext.getEnvironment(
        return Boolean.TRUE.equals(testcontainersDisabled);
```

```
    }
}
```

DatabaseContainerInitializer.java hosted with ❤ by GitHub view raw

After our container has been up and running and main database has been created, we need to create a test tenant database. For this purposes let's create **TestTenantConfig** and implement **SmartInitializingSingleton:**

```java
package com.konstde00.lab.config;

import com.konstde00.commons.domain.enums.Role;
import com.konstde00.lab.controller.AbstractApiTest;
import com.konstde00.tenant_management.domain.dto.request.CreateTenantRequestDto;
import com.konstde00.tenant_management.domain.dto.request.CreateUserRequestDto;
import com.konstde00.tenant_management.service.TenantService;
import com.konstde00.tenant_management.service.UserService;
import lombok.AccessLevel;
import lombok.RequiredArgsConstructor;
import lombok.experimental.FieldDefaults;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.dbcp2.BasicDataSource;
import org.apache.commons.lang3.StringUtils;
import org.springframework.beans.factory.SmartInitializingSingleton;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

import java.util.List;

import static com.konstde00.lab.util.DatabaseContainerInitializer.postgresMainContainer

@Slf4j
@Configuration
@RequiredArgsConstructor
@FieldDefaults(level = AccessLevel.PRIVATE, makeFinal = true)
public class TestTenantConfig implements SmartInitializingSingleton {

    UserService userService;
    TenantService tenantService;

    public static Long userId;
    public static Long adminId;

    @Override
    public void afterSingletonsInstantiated() {

        var createUserDto = CreateUserRequestDto
                .builder()
                .email("user@domain.com")
                .password("passwordOfUser")
                .build();

        var dto = CreateTenantRequestDto
                .builder()
                .name("tenant")
```

```
49                  .dbName("tenant")
50                  .dbPassword("mega_secure_password")
51                  .userName("tenant")
52                  .user(createUserDto)
53                  .build();
54
55          var createdTenant = tenantService.create(dto);
56
57          userId = createdTenant.getUserId();
58          AbstractApiTest.USER.setId(userId);
59
60          var createAdminDto = CreateUserRequestDto
61                  .builder()
62                  .email("admin@domain.com")
63                  .password("passwordOfAdmin")
64                  .tenantId(createdTenant.getId())
65                  .roles(List.of(Role.ADMIN))
66                  .build();
67
68          adminId = userService.create(createAdminDto).getId();
69          AbstractApiTest.ADMIN.setId(adminId);
70      }
71
72      @Bean(value = "tenantDataSource")
73      public DataSource tenantDataSource(){
74
75          BasicDataSource ds = new BasicDataSource();
76          ds.setUrl(postgresMainContainer.getJdbcUrl()
77                  .replace(postgresMainContainer.getDatabaseName(), StringUtils.EMPTY) +
78          ds.setUsername("tenant");
79          ds.setDriverClassName(postgresMainContainer.getDriverClassName());
80          ds.setPassword("mega_secure_password");
81          ds.setRollbackOnReturn(false);
82
83          return ds;
84      }
85  }
```

Here we have specified a flow of test tenant database creation and a tenant datasource bean, which is used in all tests in @DbRider annotation:

```
@DBRider(dataSourceBeanName = "tenantDataSource")
```

Also in some cases could be extremely important to see which SQL queries were executed. For this purpose we can use datasource proxy (**p6spy** proxy in our case) that will log queries nicely like this:

```
Some-data-point  INFO 2702 --- [           main] p6spy
insert into researches (name, description, id) values ('Demo name', 'Demo descr
```

It's enough to add required dependency, all other configuration things will be done by this library.

Hope you enjoyed this part and let me know if you have any comments or suggestions.

Summary

To summarize, I have to mention that we have created a simple demonstration of a multi-module multi-tenant application with database-per-tenant approach.

Source code of the application and simple instructions of it's launching are available in the github repository: https://github.com/konstde00/multitenancy_overview

Hope this will help you enjoy your work!

Multi Tenant | Multi Tenant Architecture | Spring Boot | Maven

Follow

Written by Kostiantyn Dementiev

103 Followers · 19 Following

Software engineer https://www.linkedin.com/in/kostiantyn-dementiev-699786223/

Responses (2)

Inetjob

What are your thoughts?

Jay Morelli
Jan 18, 2023

Very good article! Found this via the This Week In Spring blog. Thank you for the work and please keep up with the content :D

1 Reply

Lisa Shnurenko
Aug 22, 2023

👍👍

Reply

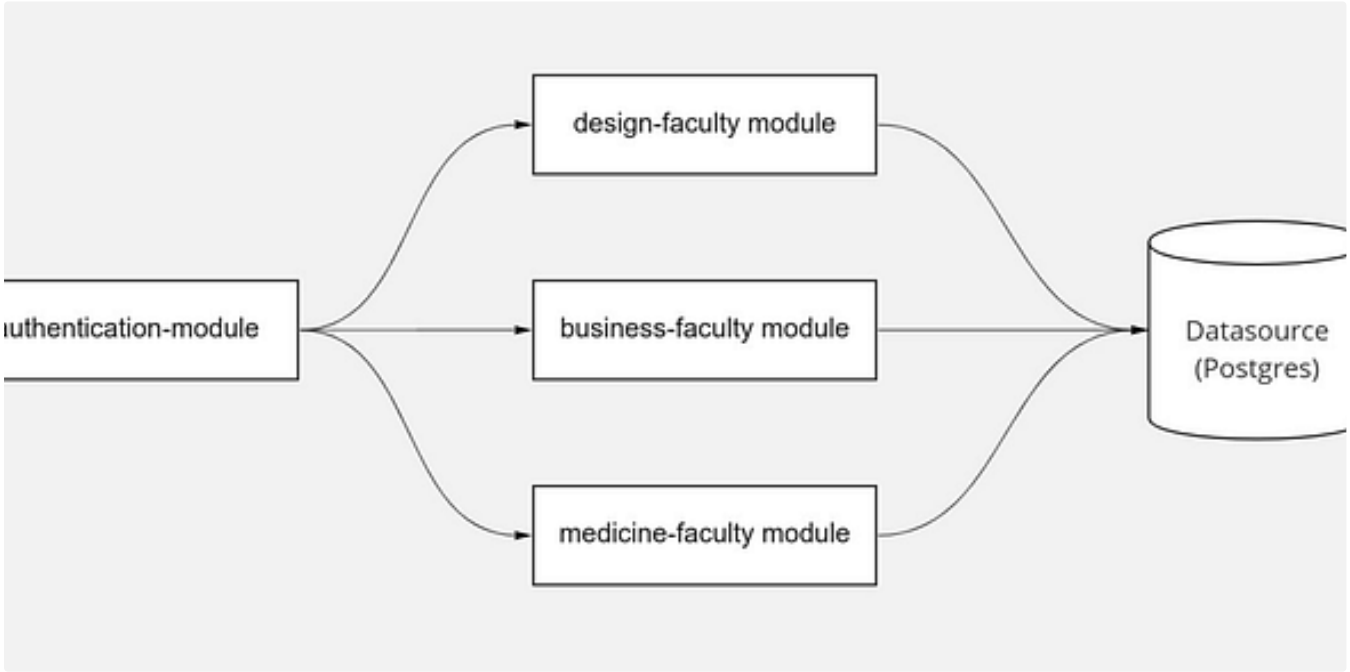
More from Kostiantyn Dementiev

Kostiantyn Dementiev

Guide to API-first development with Spring-Boot and OpenApi

In this article we will discuss developing REST APIs conforming to OpenApi Specifications and using the OpenApi code generator tool in Java...

Aug 22, 2023 36



Kostiantyn Dementiev

Getting started with Spring multi-module architecture

Developing a Spring-boot—Maven multi-module project in Intellij IDEA and supporting it can land lots of developers in great trouble.

Jun 4, 2022 26 1

Kostiantyn Dementiev

Notes about Application Security for Java developers

Welcome to the guide on keeping Java applications safe and secure! In this article, we’re going to explore why it’s super important to make...

Jan 10, 2024 15

See all from Kostiantyn Dementiev

Recommended from Medium

In ITNEXT by Saeed Zarinfam

Will GitHub Copilot Agent Mode Kill Cursor and Windsurf?

Project Padawan can be a game-changer!

Mar 2 109 2

Sithara Wanigasooriya

Best Practices and Design Patterns for Spring Boot Microservices with Maven

In this guide, I'll walk you through the best practices and design patterns for developing scalable, resilient, and maintainable...

Sep 25, 2024 57

Lists

Staff picks

822 stories · 1645 saves

Stories to Help You Level-Up at Work

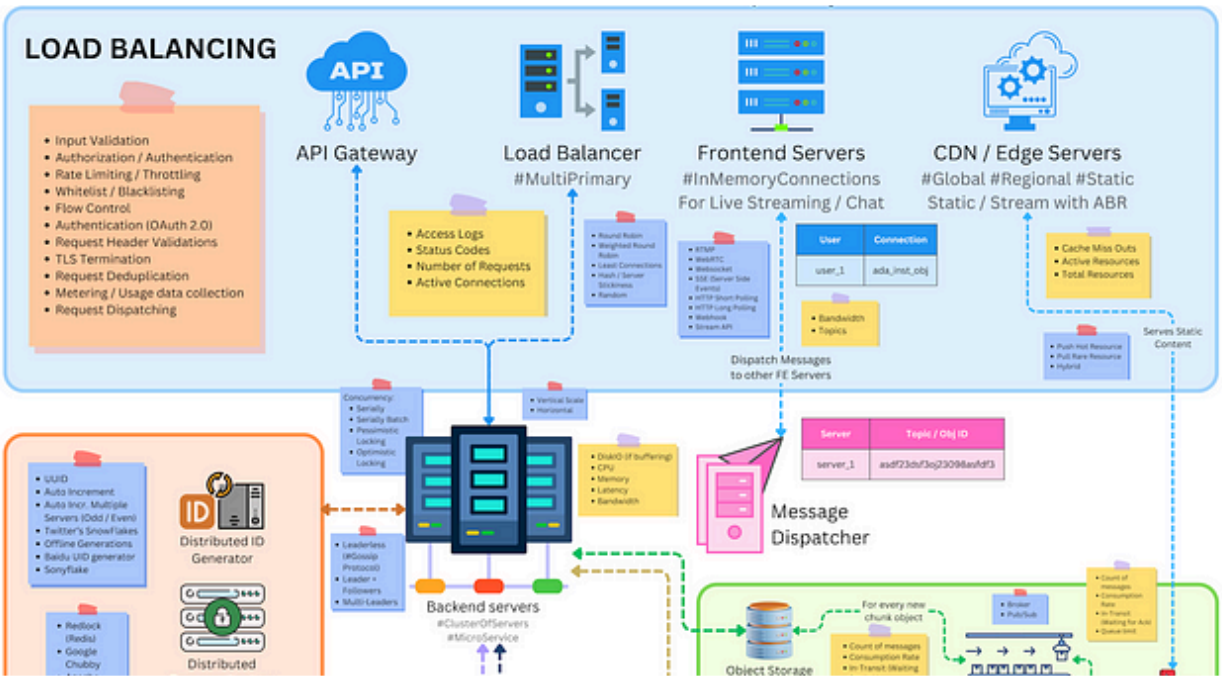
19 stories · 947 saves

Self-Improvement 101

20 stories · 3351 saves

Productivity 101

20 stories · 2815 saves



In ByteByteGo System Design Alliance by Love Sharma

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

Sep 17, 2023 9.1K 57

Ramesh Fadatare

Spring Boot CRUD Example with PostgreSQL

In this tutorial, we will build a Spring Boot CRUD (Create, Read, Update, Delete) application using PostgreSQL as the database. The...

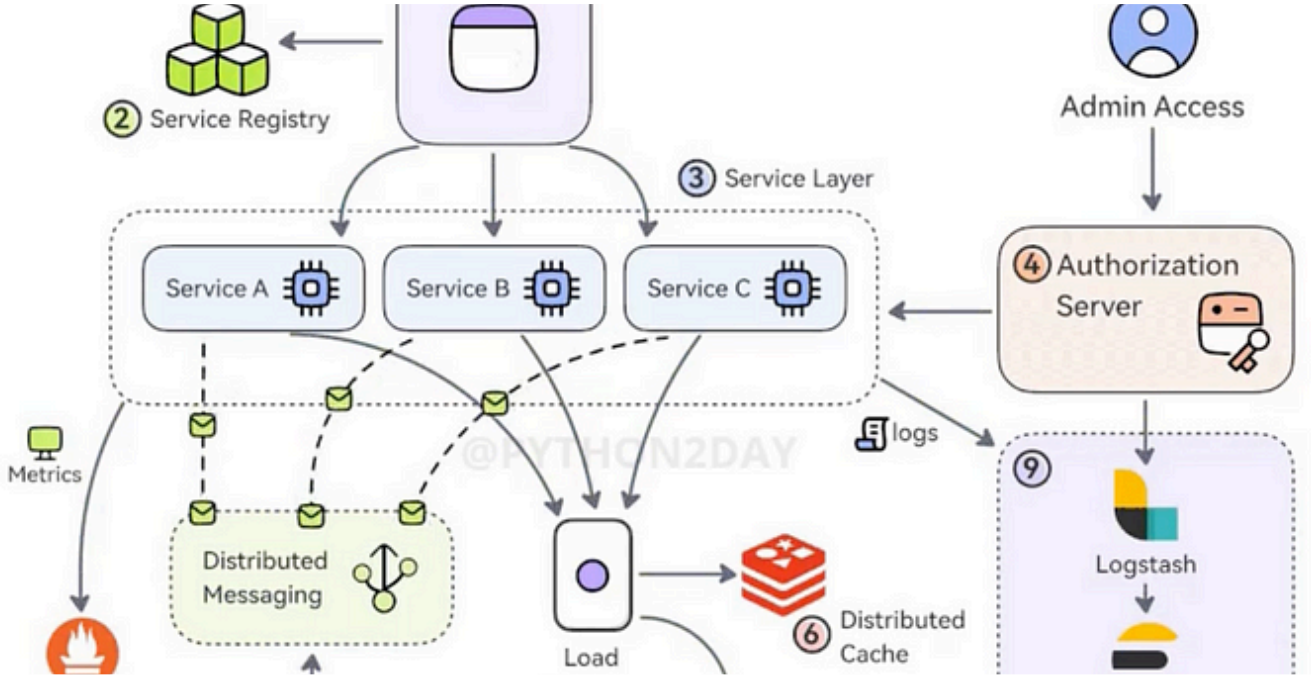
Nov 10, 2024 3

In Jeroen Rosenberg by Jeroen Rosenberg

Simplifying Hexagonal Architecture: How to Avoid Excessive Boilerplate

Hexagonal architecture, or Ports and Adapters, is a pattern that separates the core business logic of an application from its (external)...

5d ago 107 2



Roman Burdiuzha

Useful Cheat Sheet: 9 Key Components of a Production Microservices Application

Building and managing a microservices-based application requires various components that work together seamlessly. Here's a quick guide to...

Sep 17, 2024 57

See more recommendations