Spring Boot PostgreSQL CRUD

spring boot and postgessql example from scratch, i need to handle crud operaions, pagination, sorting, 1 month, 3 months and 1 year filter data

give code from scratch step by step

Here's a **complete Spring Boot + PostgreSQL example** from scratch with:

- CRUD operations
- Pagination
- Sorting
- Date-based filtering (last 1 month, 3 months, and 1 year)

✅ Tech Stack

- Spring Boot 3.x
- Spring Data JPA
- PostgreSQL
- Lombok

🔧 Step 1: Setup `pom.xml`

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>springboot-postgres-crud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>
    <!-- Spring Boot Starter -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Data JPA -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- PostgreSQL Driver -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <scope>runtime</scope>
    </dependency>

    <!-- Lombok -->
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>

    <!-- Dev tools -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

🛠 Step 2: `application.yml`

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/testdb
    username: postgres
    password: postgres
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        format_sql: true
  jackson:
    serialization:
      write-dates-as-timestamps: false
```

🧩 Step 3: Entity `UserEntity.java`

```java
package com.example.entity;

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

@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class UserEntity {

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

    private String name;
    private String email;

    private LocalDateTime createdAt;

    @PrePersist
    public void setCreationTime() {
        this.createdAt = LocalDateTime.now();
    }
}
```

📦 Step 4: Repository `UserRepository.java`

java

```
 package com.example.repository;

import com.example.entity.UserEntity;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

import java.time.LocalDateTime;

public interface UserRepository extends JpaRepository<UserEntity, Long> {

    Page<UserEntity> findByCreatedAtAfter(LocalDateTime date, Pageable pageable);
}
```

⚙️ Step 5: Service `UserService.java`

java

```
 package com.example.service;

import com.example.entity.UserEntity;
import com.example.repository.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.*;
import org.springframework.stereotype.Service;

import java.time.LocalDateTime;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;

    public UserEntity create(UserEntity user) {
        return userRepository.save(user);
    }

    public Page<UserEntity> getAll(int page, int size, String sortBy, String order, String filter) {
        Sort sort = order.equalsIgnoreCase("desc") ? Sort.by(sortBy).descending() : Sort.by(sortBy).ascending();
        Pageable pageable = PageRequest.of(page, size, sort);

        if (filter != null) {
            LocalDateTime now = LocalDateTime.now();
            LocalDateTime fromDate = switch (filter) {
                case "1month" -> now.minusMonths(1);
                case "3months" -> now.minusMonths(3);
                case "1year" -> now.minusYears(1);
                default -> null;
            };

            if (fromDate != null) {
                return userRepository.findByCreatedAtAfter(fromDate, pageable);
            }
        }

        return userRepository.findAll(pageable);
    }

    public Optional<UserEntity> getById(Long id) {
        return userRepository.findById(id);
    }
    public Optional<UserEntity> update(Long id, UserEntity user) {
        return userRepository.findById(id).map(existing -> {
            existing.setName(user.getName());
            existing.setEmail(user.getEmail());
            return userRepository.save(existing);
        });
    }

    public boolean delete(Long id) {
        return userRepository.findById(id).map(user -> {
            userRepository.delete(user);
            return true;
        }).orElse(false);
    }
}
```

🌐 Step 6: Controller `UserController.java`

```java
package com.example.controller;

import com.example.entity.UserEntity;
import com.example.service.UserService;
import lombok.RequiredArgsConstructor;
import org.springframework.data.domain.Page;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Optional;

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

    private final UserService userService;

    @PostMapping
    public ResponseEntity<UserEntity> create(@RequestBody UserEntity user) {
        return ResponseEntity.ok(userService.create(user));
    }

    @GetMapping
    public ResponseEntity<Page<UserEntity>> getAll(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "5") int size,
            @RequestParam(defaultValue = "createdAt") String sortBy,
            @RequestParam(defaultValue = "desc") String order,
            @RequestParam(required = false) String filter // values: 1month, 3months, 1year
    ) {
        return ResponseEntity.ok(userService.getAll(page, size, sortBy, order, filter));
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserEntity> getById(@PathVariable Long id) {
        return userService.getById(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PutMapping("/{id}")
    public ResponseEntity<UserEntity> update(@PathVariable Long id, @RequestBody UserEntity user) {
        return userService.update(id, user)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        return userService.delete(id)
                ? ResponseEntity.noContent().build()
                : ResponseEntity.notFound().build();
    }
}
```

📅 Example API Calls:

Endpoint	Description
`GET /api/users?page=0&size=5&sortBy=createdAt&order=desc`	Pagination & sorting
`GET /api/users?filter=1month`	Filter users created in last 1 month
`GET /api/users?filter=3months`	Filter users created in last 3 months
`GET /api/users?filter=1year`	Filter users created in last 1 year

📦 Step 7: Main Class

```java
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootPostgresCrudApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootPostgresCrudApplication.class, args);
    }
}
```

🧪 Test With Sample cURL

```bash
curl -X POST http://localhost:8080/api/users \
  -H "Content-Type: application/json" \
  -d '{"name": "John", "email": "john@example.com"}'
```

Would you like to include Swagger/OpenAPI or unit tests as well?

we can use PagingAndSortingRepository right

Yes, absolutely! You can use `PagingAndSortingRepository` instead of `JpaRepository` if your primary goals are **pagination** and **sorting**, and you **don’t need the extra features** provided by `JpaRepository` like `flush()`, `deleteInBatch()`, etc.

✅ What's the Difference?

Repository Interface	Extends	Use Case
`CrudRepository`	—	Basic CRUD
`PagingAndSortingRepository`	`CrudRepository`	CRUD + Pagination + Sorting
`JpaRepository`	`PagingAndSortingRepository`	Full JPA + Batch ops + Flush, etc.

✅ Replace `JpaRepository` with `PagingAndSortingRepository`

📦 Update Repository

```java
package com.example.repository;

import com.example.entity.UserEntity;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import java.time.LocalDateTime;
```

```
public interface UserRepository extends PagingAndSortingRepository<UserEntity, Long> {
    Page<UserEntity> findByCreatedAtAfter(LocalDateTime date, Pageable pageable);
}
```

That's it — you can continue to use all the pagination, sorting, and filtering functionality exactly the same.

🔁 If You Need `List<User>`:

`PagingAndSortingRepository` returns `Iterable<T>`, not `List<T>`. So to convert:

java

```
List<UserEntity> list = StreamSupport
    .stream(userRepository.findAll().spliterator(), false)
    .collect(Collectors.toList());
```

But if you're **only using** `Page<UserEntity>` in your service (as shown before), this won't affect you.

✅ When Should I Use `JpaRepository` Instead?

If you need any of the following:

- `findAll(Sort sort)`
- `flush()`
- `saveAllAndFlush()`
- `deleteAllInBatch()`
- JPQL or native query support from `@Query`

Then go with `JpaRepository`.

So yes — **you can safely use** `PagingAndSortingRepository` for your use case. Let me know if you want a full working version of the project with it.

what about findByCreatedAtAfter if we are using

PagingAndSortingRepository means in service

Great question!

Even when you're using `PagingAndSortingRepository`, you **can still define custom query methods** like `findByCreatedAtAfter(...)` in the repository interface. Spring Data will generate the implementation automatically for you — the **behavior does not change** whether you extend `JpaRepository` or `PagingAndSortingRepository`.

✅ Example with `PagingAndSortingRepository`:

📦 Repository:

```java
public interface UserRepository extends PagingAndSortingRepository<UserEntity, Long> {
    Page<UserEntity> findByCreatedAtAfter(LocalDateTime date, Pageable pageable);
}
```

This works perfectly and allows:

- `filter by createdAt > date`
- `pagination + sorting` using `Pageable`

⚙️ Service (No Changes Needed):

```java
public Page<UserEntity> getAll(int page, int size, String sortBy, String order, String filter) {
    Sort sort = order.equalsIgnoreCase("desc") ? Sort.by(sortBy).descending() : Sort.by(sortBy).ascending();
    Pageable pageable = PageRequest.of(page, size, sort);

    if (filter != null) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime fromDate = switch (filter) {
            case "1month" -> now.minusMonths(1);
            case "3months" -> now.minusMonths(3);
            case "1year" -> now.minusYears(1);
            default -> null;
```

```
      };

      if (fromDate != null) {
        return userRepository.findByCreatedAtAfter(fromDate, pageable);
      }
    }

    return userRepository.findAll(pageable);
  }
```

🔁 Summary:

- ✅ `findByCreatedAtAfter(...)` works with `PagingAndSortingRepository`
- ✅ `Pageable` works the same way
- ✅ No changes needed in the service or controller
- ❌ You don't get `List<T> findAll()` — use `Iterable<T>` or use a `Stream`.

Let me know if you'd like a working project zip or GitHub repo with this setup using `PagingAndSortingRepository`.

but PagingAndSortingRepository will handle sort and pagination right then why i need do my won in

Excellent point — and you're absolutely right to ask this.

You're asking:

> If `PagingAndSortingRepository` already **supports pagination and sorting**, why do I need to create `PageRequest.of(...)` manually in the service?

✅ Short Answer:

You still **need to construct a** `Pageable` **object manually (e.g., using** `PageRequest.of(...)` **)** because `PagingAndSortingRepository` **supports** pagination and sorting, but it **doesn't automatically know** the page number, size, or sort column — you **must provide** them via the `Pageable` parameter.

🧠 Think of it like this:

- `PagingAndSortingRepository` gives you a method like:

  ```java
  Page<T> findAll(Pageable pageable);
  ```

- But **you** must provide the `Pageable` instance — which includes:
  - `page number`
  - `page size`
  - `sort direction`
  - `sort field`

  via something like:

  ```java
  PageRequest.of(page, size, Sort.by("createdAt").descending())
  ```

✅ What PagingAndSortingRepository *does* for you:

It provides overloaded methods like:

```java
 Page<T> findAll(Pageable pageable);
Iterable<T> findAll(Sort sort);
```

But it **does not create** `Pageable` **objects** from request parameters — that's your job (or Spring Web does it with the right setup — more on that below 👇).

✅ Bonus Tip: Let Spring MVC Create `Pageable` Automatically

You can simplify your controller by letting Spring create the `Pageable` object from query parameters:

```java
```

```
@GetMapping
public ResponseEntity<Page<UserEntity>> getAll(Pageable pageable) {
    return ResponseEntity.ok(userRepository.findAll(pageable));
}
```

Then call:

```http
GET /api/users?page=0&size=10&sort=name,asc
```

This works **automatically** if you use `Pageable` as a parameter.

You can even customize default pagination with:

```java
@GetMapping
public ResponseEntity<Page<UserEntity>> getAll(
    @PageableDefault(size = 5, sort = "createdAt", direction = Sort.Direction.DESC) Pageable
pageable
) {
    return ResponseEntity.ok(userRepository.findAll(pageable));
}
```

✅ Summary:

Concept	Explanation
`PagingAndSortingRepository`	Gives you methods that accept a `Pageable` or `Sort`
You	Must **build the** `Pageable` **object** (manually or let Spring inject it)
Why manually create `PageRequest.of(...)`	To control page number, size, and sort — unless you're using `Pageable` directly in the controller

Concept	Explanation
Automatic injection of `Pageable`	Works if your controller method accepts `Pageable` as a parameter

---

Would you like to refactor your controller to directly use `Pageable`? I can show that version too.

then wrete time filetrs code logic

Got it! Since you're now using `PagingAndSortingRepository` and possibly allowing Spring to inject the `Pageable` object automatically, I'll show you **how to implement time-based filters** (e.g., last 1 month, 3 months, 1 year) with Spring Data JPA in a clean way.

✅ Use Case: Filter by time (createdAt) + pagination + sorting

---

✅ Step-by-step Updated Setup:

📦 Repository

You already have this (works with both `JpaRepository` and `PagingAndSortingRepository`):

```java
public interface UserRepository extends PagingAndSortingRepository<UserEntity, Long> {

    // For date-based filtering
    Page<UserEntity> findByCreatedAtAfter(LocalDateTime date, Pageable pageable);
}
```

⚙️ Service Method with Time Filter Logic

```java
public Page<UserEntity> getAllWithOptionalTimeFilter(Pageable pageable, String filter) {
    if (filter != null && !filter.isEmpty()) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime fromDate = switch (filter.toLowerCase()) {
            case "1month" -> now.minusMonths(1);
            case "3months" -> now.minusMonths(3);
            case "1year" -> now.minusYears(1);
            default -> null;
        };

        if (fromDate != null) {
            return userRepository.findByCreatedAtAfter(fromDate, pageable);
        }
    }

    return userRepository.findAll(pageable); // No time filter applied
}
```

🌐 Controller (Use Pageable directly)

```java
@GetMapping
public ResponseEntity<Page<UserEntity>> getAll(
        @RequestParam(required = false) String filter, // 1month, 3months, 1year
        Pageable pageable
) {
    Page<UserEntity> result = userService.getAllWithOptionalTimeFilter(pageable, filter);
    return ResponseEntity.ok(result);
}
```

✅ Example Requests:

1. **Pagination only**

   `GET /api/users?page=0&size=5&sort=name,asc`

2. **Filter: Last 1 Month**

   `GET /api/users?filter=1month&page=0&size=10&sort=createdAt,desc`

3. **Filter: Last 3 Months with sorting**

   `GET /api/users?filter=3months&sort=name,asc`

4. **Filter: Last 1 Year**

   `GET /api/users?filter=1year`

✅ Bonus: Custom Pageable Defaults

You can use this if you want to apply default paging/sorting in controller:

```java
@GetMapping
public ResponseEntity<Page<UserEntity>> getAll(
        @RequestParam(required = false) String filter,
        @PageableDefault(size = 5, sort = "createdAt", direction = Sort.Direction.DESC) Pageable pageable
) {
    return ResponseEntity.ok(userService.getAllWithOptionalTimeFilter(pageable, filter));
}
```

✅ Summary

Feature	Handled
Pagination	✅ via `Pageable`
Sorting	✅ via `Pageable` and `sort` query param
Time Filter (1m, 3m, 1y)	✅ via `filter` param and `createdAt` logic
Repository	✅ still `PagingAndSortingRepository`

Let me know if you want this with Swagger, OpenAPI, or integrated test cases.