📌 Step 1: What is Unpoly?

1.1 What is Unpoly?

Unpoly is a JavaScript framework that helps update parts of a webpage **without refreshing the whole page**.
It is similar to **HTMX** and **Laravel Livewire**, but with **more features**.

1.2 Why Use Unpoly?

✅ **No full-page reloads** – Update only parts of the page
✅ **Works with server-side rendering** – Perfect for **Spring Boot + Thymeleaf**
✅ **Dynamic forms and tables** – Like Laravel Livewire
✅ **Modals, caching, and navigation** – Built-in features

1.3 How Does Unpoly Work?

- Instead of refreshing the whole page, Unpoly **updates only specific sections**.
- You add simple `up-*` attributes to **links, buttons, and forms** to enable Unpoly.
- Example:

```
<a up-target="#content" href="/about">Load About Page</a>
```

This link **loads the About page inside** the `#content` div **without refreshing**.

📌 Step 2: Setting Up Unpoly

2.1 Install Unpoly in Your Project

To use Unpoly, **add this to your HTML <head> section**:

```
<head>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
```

That's it! Now, Unpoly is ready to use.

2.2 First Example: Loading Content Without Reload

Let's create a simple web page where clicking a button **loads new content without refreshing**.

index.html (Main Page)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly Example</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
</head>
<body>
    <h2>Welcome to Unpoly</h2>

    <!-- Content will be updated here -->
    <div id="content">
        <p>Click the button to load new content without refreshing.</p>
    </div>

    <!-- Button to load new content -->
    <button up-target="#content" up-follow href="new-content.html">Load New Content</button>
</body>
</html>
```

new-content.html (New Content)

```
<p>This is the new content loaded with Unpoly! 🚀</p>
```

2.3 How Does It Work?

1. The button has up-target="#content", which tells Unpoly to **update only** the #content div.
2. Clicking the button **loads new-content.html into the #content div**.
3. No full-page reload happens!

🎯 **Result**: The content changes instantly without refreshing the page.

✅ Step 2 Summary

- **Unpoly works by updating only parts of a webpage**.
- **You need to add Unpoly to your <head>**.
- **Use up-target to replace content dynamically**.

📌 Step 3: Handling Forms Without Reload

In this step, you will learn how to:

✅ **Submit forms without reloading the page**
✅ **Update only specific parts of the page**
✅ **Show form validation messages dynamically**

3.1 Basic Form Submission with Unpoly

Let's create a simple form where the user enters their **name** and **email**, and the submitted data is displayed **without refreshing the page**.

📝 Backend (Spring Boot)

First, create a simple **Spring Boot Controller** to handle form submissions.

📌 UserController.java

```
package com.example.unpoly_demo.controller;

import lombok.AllArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@Controller
@AllArgsConstructor
@RequestMapping("/users")
public class UserController {
    private final List<User> users = new ArrayList<>();

    // Show the form page
    @GetMapping
    public String showForm(Model model) {
        model.addAttribute("users", users);
        return "users/form";
    }

    // Handle form submission
    @PostMapping("/add")
    public String addUser(@ModelAttribute User user, Model model) {
        users.add(user);  // Add the user to the list
        model.addAttribute("users", users);
        return "users/table :: userTable"; // Return only the table part
    }
}
```

📝 Thymeleaf Templates

📌 form.html (Main Page)

This page contains:

✅ **A form for adding users**
✅ **A table that updates dynamically**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly Form Example</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <h2>Add a User</h2>

    <!-- User Form -->
    <form up-submit="users/add" up-target="#userTable">
        <input type="text" name="name" placeholder="Enter Name" required>
        <input type="email" name="email" placeholder="Enter Email" required>
        <button type="submit">Add User</button>
    </form>

    <h2>User List</h2>
    <!-- Table will be updated dynamically -->
    <div id="userTable">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                </tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

📌 **table.html (Table Fragment)**

This file contains only the **table** that updates dynamically.

```
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody id="userTable">
        <tr th:each="user : ${users}">
            <td th:text="${user.name}"></td>
            <td th:text="${user.email}"></td>
        </tr>
    </tbody>
</table>
```

3.2 How Does It Work?

✅ The form has up-submit="users/add"
→ This tells Unpoly to **send the form data without reloading the page**.

✅ The form also has up-target="#userTable"
→ This means **only the table updates** when a new user is added.

✅ The controller **returns only the table fragment**
→ This ensures that only the required part of the page is updated.

🎯 **Result:**

- When you **submit the form**, the new user appears in the table **without refreshing** the page.
- **Only the table updates**, making it **super fast**!

3.3 Adding Form Validation

Now, let's add **error validation** for the form.

Modify the UserController.java:

📌 UserController.java (Updated)

```
@PostMapping("/add")
public String addUser(@ModelAttribute User user, Model model) {
    if (user.getName().isEmpty() || user.getEmail().isEmpty()) {
        model.addAttribute("error", "Name and Email are required!");
        return "users/form :: errorMessage";
    }
    users.add(user);
    model.addAttribute("users", users);
    return "users/table :: userTable";
}
```

📌 Update form.html to Show Errors

Modify the form to display error messages dynamically.

```
<!-- Error message (hidden by default) -->
<div id="errorMessage" style="color: red;" up-if-error>
    <p th:text="${error}"></p>
</div>

<!-- User Form -->
<form up-submit="users/add" up-target="#userTable" up-fail-target="#errorMessage">
    <input type="text" name="name" placeholder="Enter Name" required>
    <input type="email" name="email" placeholder="Enter Email" required>
    <button type="submit">Add User</button>
</form>
```

🎯 Step 3 Summary

✅ **Form submits without refreshing the page**
✅ **Only the table updates dynamically**
✅ **Error messages appear dynamically**

📌 Step 4: Dynamic Table Filtering with Unpoly

We'll create a **user list** that can be filtered dynamically by **name** and **email**.

4.1 Backend (Spring Boot Controller)

Modify the controller to handle filtering.

📌 UserController.java

```
package com.example.unpoly_demo.controller;

import lombok.AllArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@Controller
@AllArgsConstructor
public class UserController {
    private final List<User> users = new ArrayList<>();

    // Initialize some users (for demo)
    public void initUsers() {
        users.add(new User("Alice", "alice@example.com"));
        users.add(new User("Bob", "bob@example.com"));
        users.add(new User("Charlie", "charlie@example.com"));
    }

    // Show user list page
    @GetMapping("/users")
    public String showUsers(Model model) {
        initUsers();
        model.addAttribute("users", users);
        return "users/list";
    }
```

```
    // Filter users dynamically
    @GetMapping("/users/filter")
    public String filterUsers(
        @RequestParam(required = false) String name,
        @RequestParam(required = false) String email,
        Model model
    ) {
        List<User> filteredUsers = users;

        // Filter by name
        if (name != null && !name.isEmpty()) {
            filteredUsers = filteredUsers.stream()
                    .filter(user -> user.getName().toLowerCase().contains(name.toLowerCase()))
                    .collect(Collectors.toList());
        }

        // Filter by email
        if (email != null && !email.isEmpty()) {
            filteredUsers = filteredUsers.stream()
                    .filter(user -> user.getEmail().toLowerCase().contains(email.toLowerCase()))
                    .collect(Collectors.toList());
        }

        model.addAttribute("users", filteredUsers);
        return "users/table :: userTable"; // Return only the table fragment
    }
}
```

4.2 Thymeleaf Templates

📌 list.html (Main Page with Filters & Table)

This page contains:

✅ **Filter inputs**
✅ **Table that updates dynamically**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly Filtering Example</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
```

```
</head>
<body>
    <h2>Dynamic User Filtering</h2>

    <!-- Filter Form -->
    <form up-change="users/filter" up-target="#userTable">
        <input type="text" name="name" placeholder="Filter by Name">
        <input type="email" name="email" placeholder="Filter by Email">
    </form>

    <h2>User List</h2>
    <!-- Table will be updated dynamically -->
    <div id="userTable">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                </tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

📌 table.html (Table Fragment)

This file contains only the **table** that updates dynamically.

```
<table>
    <thead>
        <tr>
            <th>Name</th>
            <th>Email</th>
        </tr>
    </thead>
    <tbody id="userTable">
        <tr th:each="user : ${users}">
            <td th:text="${user.name}"></td>
            <td th:text="${user.email}"></td>
```

```
        </tr>
    </tbody>
</table>
```

4.3 How It Works

✅ The filter inputs have up-change="users/filter" up-target="#userTable"
→ When you **type in the fields**, Unpoly **sends the filter request automatically**.

✅ The **server filters the users** based on input values and returns **only the table fragment**.

✅ The table **updates dynamically without refreshing** the whole page.

🎯 **Result:**

- The table **updates instantly** as you type the filter values.
- Only the **filtered users** appear in the table.
- **No page reload happens!**

🎯 Step 4 Summary

✅ **Unpoly allows real-time table filtering**
✅ **Only the table updates dynamically**
✅ **Filter results appear instantly without refreshing the page**

📌 Step 5: Using Unpoly Modals for Editing & Deleting

We'll add:

- A **"Edit" button** to update user details inside a modal.
- A **"Delete" button** to remove users dynamically.

5.1 Backend (Spring Boot Controller)

Modify UserController.java to handle **editing and deleting users**.

📌 **UserController.java**

```
package com.example.unpoly_demo.controller;

import lombok.AllArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Optional;

@Controller
@AllArgsConstructor
@RequestMapping("/users")
public class UserController {
    private final List<User> users = new ArrayList<>();

    // Initialize users
    public void initUsers() {
        if (users.isEmpty()) {
            users.add(new User(1, "Alice", "alice@example.com"));
            users.add(new User(2, "Bob", "bob@example.com"));
            users.add(new User(3, "Charlie", "charlie@example.com"));
        }
    }

    @GetMapping
    public String showUsers(Model model) {
        initUsers();
        model.addAttribute("users", users);
        return "users/list";
    }

    // Show edit form in a modal
    @GetMapping("/edit/{id}")
    public String showEditForm(@PathVariable int id, Model model) {
        Optional<User> user = users.stream().filter(u -> u.getId() == id).findFirst();
        user.ifPresent(value -> model.addAttribute("user", value));
        return "users/edit-form :: editModal";
    }
```

```
    // Handle edit form submission
    @PostMapping("/update")
    public String updateUser(@ModelAttribute User updatedUser, Model model) {
        for (User user : users) {
            if (user.getId() == updatedUser.getId()) {
                user.setName(updatedUser.getName());
                user.setEmail(updatedUser.getEmail());
            }
        }
        model.addAttribute("users", users);
        return "users/table :: userTable";
    }

    // Delete user
    @PostMapping("/delete/{id}")
    public String deleteUser(@PathVariable int id, Model model) {
        users.removeIf(user -> user.getId() == id);
        model.addAttribute("users", users);
        return "users/table :: userTable";
    }
}
```

5.2 Thymeleaf Templates

📌 list.html (Main Page with Table & Buttons)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly Modals Example</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <h2>User Management</h2>

    <!-- User List -->
    <div id="userTable">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
```

```
                <th>Email</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.name}"></td>
                <td th:text="${user.email}"></td>
                <td>
                    <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
                        Edit
                    </button>
                    <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable">
                        Delete
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
</div>

<!-- Modal Placeholder -->
<div id="editModal"></div>
</body>
</html>
```

📌 edit-form.html (Edit Form Modal)

```
<div id="editModal" class="modal">
    <h2>Edit User</h2>
    <form up-submit="users/update" up-target="#userTable">
        <input type="hidden" name="id" th:value="${user.id}">
        <input type="text" name="name" th:value="${user.name}" required>
        <input type="email" name="email" th:value="${user.email}" required>
        <button type="submit">Save Changes</button>
        <button up-dismiss>Cancel</button>
    </form>
</div>
```

5.3 How It Works

✅ The **"Edit" button** has up-layer="modal" up-target="#editModal"
→ This opens the **edit form in a modal** dynamically.

✅ The **form inside the modal** submits with up-submit="users/update" up-target="#userTable"
→ The table updates instantly when the form is submitted.

✅ The **"Delete" button** has up-submit="users/delete/{id}" up-target="#userTable"
→ This deletes the user **without refreshing** the page.

🎯 **Result:**

- Clicking "Edit" **opens a modal** with the user's details.
- Editing and submitting **updates the table instantly**.
- Clicking "Delete" **removes the user dynamically**.

🎯 Step 5 Summary

✅ **Modals allow editing data dynamically**
✅ **Forms submit without refreshing**
✅ **Deleting records updates only the table**

📌 Step 6: Adding Loading Indicators & Toast Messages

6.1 Adding a Loading Indicator

📌 Loading Indicator in Unpoly

Unpoly provides a simple way to show a loading spinner while content is being fetched.

📌 list.html (Main Page with Loading Indicator)

You can use up-loading to show a loading spinner when the page is fetching data.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly with Loading Indicator</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <h2>User Management</h2>

    <!-- Loading Indicator -->
    <div id="loadingIndicator" style="display: none;">
        <p>Loading...</p> <!-- You can replace this with a spinner -->
    </div>

    <!-- User List -->
    <div id="userTable" up-loading="true">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th>Actions</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                    <td>
                        <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
                            Edit
                        </button>
                        <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable">
                            Delete
                        </button>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Explanation:

- **up-loading="true"**: This is used to show a **loading spinner** while content is being fetched.
- You can replace the div inside the #loadingIndicator with any custom spinner or animation to give the user a visual cue that content is being loaded.

6.2 Toast Messages for Feedback

📌 Toast Message Concept

A **toast message** is a small notification that appears temporarily to inform the user about the result of an action (success, error, etc.).

We'll show a success message after a user is added or updated.

📌 list.html (Main Page with Toast Notification)

First, add a simple **toast container** that will hold the success or error messages:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Unpoly with Toast Notifications</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <h2>User Management</h2>

    <!-- Toast Container -->
    <div id="toastContainer"></div>

    <!-- User List -->
    <div id="userTable" up-loading="true">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
```

```
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.name}"></td>
                <td th:text="${user.email}"></td>
                <td>
                    <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
                        Edit
                    </button>
                    <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable" up-
redirect="false">
                        Delete
                    </button>
                </td>
            </tr>
        </tbody>
    </table>
  </div>
</body>
</html>
```

Now, we will modify the **Spring Boot Controller** to include success messages after certain actions (like adding or deleting users).

📌 UserController.java (Adding Toast Success Messages)

When a user is added or deleted, we will add a **toast message** to indicate success.

Updated Controller Code

```
@PostMapping("/update")
public String updateUser(@ModelAttribute User updatedUser, Model model) {
    for (User user : users) {
        if (user.getId() == updatedUser.getId()) {
            user.setName(updatedUser.getName());
            user.setEmail(updatedUser.getEmail());
        }
    }
    model.addAttribute("users", users);
    model.addAttribute("toast", "User updated successfully!");
    return "users/table :: userTable";
}
```

```
@PostMapping("/delete/{id}")
public String deleteUser(@PathVariable int id, Model model) {
    users.removeIf(user -> user.getId() == id);
    model.addAttribute("users", users);
    model.addAttribute("toast", "User deleted successfully!");
    return "users/table :: userTable";
}
```

📌 Adding Toast to the Table Template

Now, in the **table.html**, we will display the **toast notification** dynamically using Unpoly.

```
<!-- Toast Message (This will be updated dynamically) -->
<div id="toastContainer" up-append>
    <p th:text="${toast}" style="background-color: #4CAF50; color: white; padding: 10px; margin: 10px;">
        <!-- Success Message -->
    </p>
</div>
```

📌 How It Works

✅ **Loading Indicator**:

- **up-loading="true"**: This attribute shows the loading indicator when the page is fetching data (like adding or deleting a user).

✅ **Toast Message**:

- The toast message is **dynamically added to the page** via model.addAttribute("toast", "Message here") in the controller.
- We append the message to the #toastContainer with the up-append attribute.

🎯 **Result:**

- When you add, edit, or delete a user, you'll see:
    - **A loading spinner** while the operation is processing.
    - **A toast message** showing success or error after the operation.

🎯 Step 6 Summary

✅ **Loading indicator** shows while content is being fetched.
✅ **Toast messages** provide real-time feedback to the user (success or failure).

📌 Step 7: Optimizing Unpoly for Advanced Use Cases

7.1 Caching for Faster Page Loads

One way to optimize Unpoly and improve the user experience is to cache the parts of the page that don't change often. This way, Unpoly can reuse previously fetched content, making your app feel faster.

📌 Caching with `up-cache`

You can use the `up-cache` attribute to specify regions of the page that should be cached.

Example: Caching the User List Table

You might want to cache the user list table so that it doesn't need to be fetched repeatedly when the user navigates back to the list.

📌 `list.html` (Main Page with Caching)

```
<div id="userTable" up-cache="true" up-loading="true">
  <table>
    <thead>
      <tr>
        <th>Name</th>
        <th>Email</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="user : ${users}">
        <td th:text="${user.name}"></td>
```

```
            <td th:text="${user.email}"></td>
            <td>
                <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
                    Edit
                </button>
                <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>
</div>
```

How It Works:

- **up-cache="true"**: This tells Unpoly to **cache the content of the #userTable**.
- When the table is re-rendered (such as after editing a user), Unpoly will reuse the cached version if available, speeding up the process.

7.2 Handling Multiple Dynamic Regions

In a more complex page, you may have several dynamic regions (like a sidebar or nested content). Unpoly can help you manage these regions efficiently.

For example, you might have a **sidebar** that filters users by various categories (e.g., "Active", "Inactive", "Admin").

📌 list.html with Multiple Dynamic Regions

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Advanced Unpoly Example</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <h2>User Management</h2>
```

```
  <!-- Sidebar Filter -->
  <div id="sidebar">
    <form up-change="users/filter" up-target="#userTable">
      <label for="status">Filter by Status:</label>
      <select name="status">
        <option value="">All</option>
        <option value="active">Active</option>
        <option value="inactive">Inactive</option>
        <option value="admin">Admin</option>
      </select>
    </form>
  </div>

  <!-- User List Table -->
  <div id="userTable" up-cache="true">
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Email</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <tr th:each="user : ${users}">
          <td th:text="${user.name}"></td>
          <td th:text="${user.email}"></td>
          <td>
            <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
              Edit
            </button>
            <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable">
              Delete
            </button>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</body>
</html>
```

Explanation:

- The **sidebar** uses up-change="users/filter" up-target="#userTable" to dynamically filter the users.
- **Both the sidebar** and **the table** are treated as **dynamic regions** that can be independently updated without full page reloads.

7.3 Using up-replace and up-append for Dynamic Content Updates

Unpoly provides powerful content replacement and appending features to make updates more dynamic.

- **up-replace**: Replaces the target content with new content.
- **up-append**: Appends new content to the target element.

📌 Example: Appending New Users

When you add a new user, you may want to append them to the table instead of replacing the entire table.

```
<form up-submit="users/add" up-target="#userTable" up-append>
   <input type="text" name="name" placeholder="Name" required>
   <input type="email" name="email" placeholder="Email" required>
   <button type="submit">Add User</button>
</form>
```

How It Works:

- **up-append**: This attribute tells Unpoly to **append the new user** to the existing table rather than replacing it.
- The **table grows dynamically**, and the page content doesn't need to be reloaded entirely.

7.4 Handling Modal Form Submissions Dynamically

When using modals for forms (like adding or editing users), it’s important to handle form submissions dynamically.

You can update specific parts of the page (e.g., a table or list) after a modal form submission.

📌 Example: Submit Form in Modal

In the previous examples, when editing a user in the modal, the form submits and updates the table:

```
<!-- Modal Form -->
<form up-submit="users/update" up-target="#userTable" up-replace>
    <input type="hidden" name="id" th:value="${user.id}">
    <input type="text" name="name" th:value="${user.name}" required>
    <input type="email" name="email" th:value="${user.email}" required>
    <button type="submit">Save Changes</button>
</form>
```

How It Works:

- **up-replace**: This tells Unpoly to **replace** the #userTable with the updated table after the form is submitted.

🎯 Step 7 Summary

✅ **Caching** speeds up the page by reusing previously fetched content.
✅ **Multiple dynamic regions** (like sidebars and tables) can be handled independently.
✅ **up-replace** and **up-append** provide fine-grained control over how content is updated.
✅ **Modal forms** can submit and dynamically update the page.

📌 Step 8: Advanced Unpoly Features & Optimization for Performance

8.1 Optimizing Page Transitions

By default, Unpoly provides **smooth transitions** between page states. This ensures that content updates don't appear abrupt, providing a seamless user experience.

However, we can take it further by using **page transition options** to optimize performance, especially on larger pages.

📌 Using `up-transition` for Smooth Page Transitions

You can control how Unpoly transitions between different page parts. Let's say we want to add a fade effect when content is updated.

Example: Adding Fade Transition to Content Updates

```
<div id="userTable" up-cache="true" up-transition="fade" up-loading="true">
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.name}"></td>
                <td th:text="${user.email}"></td>
                <td>
                    <button up-layer="modal" up-target="#editModal" th:attr="up-
follow=@{/users/edit/{id}(id=${user.id})}">
                        Edit
                    </button>
                    <button up-submit="@{/users/delete/{id}(id=${user.id})}" up-target="#userTable">
                        Delete
                    </button>
                </td>
            </tr>
        </tbody>
```

```
    </table>
</div>
```

How It Works:

- **up-transition="fade"**: This applies a fade-in transition effect whenever content is updated.
- This makes the page feel smoother and more natural when data is being updated, rather than just changing instantly.

8.2 Progressive Enhancement

Progressive enhancement ensures that users on slower networks or with limited browsers still get a usable experience, even if they don't benefit from all Unpoly's features.

You can ensure that **Unpoly's AJAX-based features** are **only applied when available**, and fallback to traditional navigation when necessary.

📌 Example: Enabling Unpoly Only When Supported

Unpoly provides a built-in mechanism for progressive enhancement by using the **unpoly.js** script in your project. It automatically degrades gracefully if JavaScript is disabled.

Example: Check for Unpoly Support

In the head section of your HTML, you can ensure that Unpoly is only used when supported:

```
<script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
<script>
    if (typeof up !== 'undefined') {
        // Unpoly is supported, initialize it
        up.start();
    } else {
        // Fallback to standard navigation
    }
</script>
```

How It Works:

- The **if (typeof up !== 'undefined')** check ensures that Unpoly's AJAX features are only initialized when the browser supports JavaScript.
- If Unpoly is unavailable (e.g., if JavaScript is disabled), the app will still work normally, using traditional navigation.

8.3 Background Fetching for Better User Experience

Unpoly allows you to **preload** content in the background so that users don't have to wait for it when they navigate to different parts of your app.

📌 Example: Preload User Details in the Background

Let's say you have a user profile page that can be dynamically loaded. You can start fetching it in the background before the user clicks on it.

```
<button up-follow="@{/users/{id}}?up-load=background" th:attr="up-follow=@{/users/{id}(id=${user.id})}">
   View Profile
</button>
```

How It Works:

- **up-load=background**: This tells Unpoly to **fetch the content in the background**. When the user clicks on the link, the content will already be available, making the page load instantly.
- This is particularly useful for applications with **long loading times** for certain content.

8.4 Lazy Loading for Images and Heavy Resources

One of the key performance optimization techniques is **lazy loading** for images and other heavy resources. This technique ensures that **resources are only loaded when they are visible on the screen**.

Unpoly doesn't directly provide lazy loading for images, but you can integrate it with the **native HTML loading="lazy" attribute** or use a JavaScript library like lazysizes.

📌 Example: Lazy Load Images Using HTML Native Feature

```
<img src="path/to/image.jpg" loading="lazy" alt="User Image">
```

How It Works:

- **loading="lazy"**: This native HTML attribute tells the browser to load images only when they come into the user's viewport (visible on screen), reducing the page's initial load time and improving performance.

8.5 Optimizing for SEO and Accessibility

While Unpoly is focused on client-side rendering, it's important to ensure that your app is still **SEO-friendly** and accessible to all users.

📌 Optimizing for SEO with Unpoly

To ensure your content is indexed by search engines and accessible to screen readers, you can **use server-side rendering** (SSR) or include **fallbacks for key content**.

Example: Fallback Content for SEO

You can use Unpoly's up-restore to ensure that when content is loaded dynamically, it's still accessible to search engines.

```
<div id="userTable" up-restore>
    <!-- Table content -->
</div>
```

How It Works:

- **up-restore**: This ensures that any content dynamically loaded via Unpoly is still available for crawlers and users with JavaScript disabled.

🎯 Step 8 Summary

✅ **Smooth page transitions** make updates appear seamless.
✅ **Progressive enhancement** ensures accessibility and performance across all devices.
✅ **Background fetching** improves load times for dynamic content.
✅ **Lazy loading** reduces the initial page load time by loading images only when necessary.
✅ **SEO and accessibility** optimization ensures that your site remains crawlable and usable by all users.

📌 Step 9: Best Practices for Deploying Unpoly Apps

9.1 Optimizing Unpoly Assets for Production

Before deploying your app, it's essential to **optimize** all assets (JavaScript, CSS, and images) to improve load times and reduce bandwidth usage.

📌 Minifying JavaScript and CSS

In production, you should **minify** your JavaScript and CSS files to reduce their size.

- **Unpoly** comes with its own minified version, but you might also want to **bundle** and **minify** your custom JavaScript and CSS using build tools like **Webpack** or **Maven**.

Example: Minifying with Maven

If you're using Maven with Spring Boot, you can use a plugin like frontend-maven-plugin to automatically handle the minification of assets during your build process.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>frontend-maven-plugin</artifactId>
```

```
    <version>1.12</version>
    <configuration>
        <nodeVersion>v14.17.0</nodeVersion>
        <npmVersion>7.20.0</npmVersion>
        <workingDirectory>src/main/frontend</workingDirectory>
    </configuration>
</plugin>
```

You can configure **Webpack** to bundle your JavaScript and CSS into minified files before deployment.

How It Works:

- By **minifying and bundling** your assets, you can significantly reduce the page load time, especially for users on slower networks.

9.2 Configuring Server-Side Rendering (SSR) with Spring Boot

In some scenarios, you might want to enable **server-side rendering** (SSR) for your dynamic content. SSR can improve performance, particularly for SEO and the initial page load.

You can achieve this by using **Thymeleaf** templates and serving dynamic content with **Spring Boot** as the backend, while still using Unpoly to handle the dynamic updates on the client side.

📌 Example: Setting up SSR in Spring Boot

In your Spring Boot project, you can serve **Thymeleaf templates** for the initial page render and use **Unpoly** for subsequent dynamic content updates.

Controller Example:

```
@Controller
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
```

```
    public String listUsers(Model model) {
        List<User> users = userRepository.findAll();
        model.addAttribute("users", users);
        return "user-list";
    }

    @PostMapping("/users/add")
    public String addUser(@ModelAttribute User user, Model model) {
        userRepository.save(user);
        return "user-list :: userTable"; // Return updated table fragment
    }
}
```

Thymeleaf Template (user-list.html):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>User List</title>
    <script src="https://unpkg.com/unpoly@2.7.2/unpoly.min.js"></script>
    <link rel="stylesheet" href="https://unpkg.com/unpoly@2.7.2/unpoly.min.css">
</head>
<body>
    <div id="userTable" up-cache="true">
        <table>
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                </tr>
            </thead>
            <tbody>
                <tr th:each="user : ${users}">
                    <td th:text="${user.name}"></td>
                    <td th:text="${user.email}"></td>
                </tr>
            </tbody>
        </table>
    </div>

    <form up-submit="users/add" up-target="#userTable" up-append>
        <input type="text" name="name" placeholder="Name">
        <input type="email" name="email" placeholder="Email">
        <button type="submit">Add User</button>
    </form>
</body>
</html>
```

How It Works:

- **Thymeleaf templates** are used for the initial page load.
- After submitting a form (like adding a user), the content is dynamically updated via Unpoly with a **fragment** (:: userTable).
- This combination of **SSR** and **Unpoly** ensures that your app is SEO-friendly and performant.

9.3 Setting up Caching Strategies

Caching can improve the performance of your Unpoly app by reducing the need to request data repeatedly. There are different ways to implement caching in Spring Boot, including **HTTP caching** and **server-side caching**.

📌 Example: Caching in Spring Boot

In Spring Boot, you can use the @Cacheable annotation to cache the results of a method. This helps reduce database load and speeds up data retrieval.

Example: Caching User Data

```
@Service
public class UserService {

    @Cacheable("users")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
}
```

How It Works:

- **@Cacheable("users")**: The first time getAllUsers() is called, the result is cached.
- The next time the method is called, Spring retrieves the result from the cache, improving performance.

Browser-side Caching for Unpoly

To further enhance performance, you can instruct the browser to **cache** certain assets, like JavaScript, CSS, and images. This can be done by setting appropriate **HTTP headers**.

For example, you can set cache headers in your Spring Boot app:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/static/**")
                .addResourceLocations("classpath:/static/")
                .setCachePeriod(3600); // Cache for 1 hour
    }
}
```

How It Works:

- This **caches static assets** for a set period of time (in this case, 1 hour), reducing the number of requests made to the server.

9.4 Ensuring Security for Unpoly-Powered Applications

Security is critical for any web application. When using Unpoly, you should take several measures to protect your app from common security vulnerabilities.

📌 Cross-Site Request Forgery (CSRF) Protection

Spring Security provides built-in protection against **CSRF attacks**, and you should ensure that your forms are protected.

Enabling CSRF Protection in Spring Boot

Spring Boot enables CSRF protection by default. You can use a token to protect Unpoly form submissions.

```
<form up-submit="@{/users/add}" method="post">
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
    <!-- Form fields here -->
    <button type="submit">Add User</button>
</form>
```

How It Works:

- The **CSRF token** is automatically included in the form to prevent CSRF attacks.

9.5 Monitoring and Performance Optimization

After deploying your app, it's important to monitor its performance and optimize it as necessary.

📌 Using Spring Boot Actuator for Monitoring

Spring Boot provides the **Actuator** module for monitoring the health of your application.

Example: Enabling Spring Boot Actuator

xml

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

Once enabled, you can access several useful endpoints to monitor the app:

- `/actuator/health`: Check if the app is healthy.
- `/actuator/metrics`: Get detailed performance metrics.

How It Works:

- **Spring Boot Actuator** helps you monitor key metrics like request times, error rates, and memory usage.

🎯 Step 9 Summary

✅ **Optimizing Unpoly assets** by minifying and bundling files improves performance.
✅ **SSR with Spring Boot** can enhance SEO and initial page load performance.
✅ **Caching strategies** can significantly speed up repeated requests and data retrieval.
✅ **Security**: Always protect forms with CSRF tokens to prevent attacks.
✅ **Monitoring and performance** tools like **Spring Boot Actuator** help you ensure the app runs smoothly after deployment.

📌 Step 10: Scaling Unpoly Applications for High Traffic

10.1 Scaling Horizontally to Handle More Traffic

Horizontal scaling involves adding more instances of your application to distribute traffic. This is particularly important when your app is experiencing high traffic and a single server cannot handle the load.

📌 Example: Deploying Spring Boot in a Cluster

To horizontally scale your Spring Boot app, you can deploy it on multiple instances (e.g., in a **cloud environment** like AWS, Google Cloud, or Azure).

1. **Containerization with Docker**: Use Docker to containerize your Spring Boot application so it can be easily replicated and deployed across multiple servers or instances.

Dockerfile Example:

Dockerfile

```
FROM openjdk:11-jre-slim
EXPOSE 8080
ADD target/myapp.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

2. **Orchestration with Kubernetes**: Kubernetes can manage the scaling and deployment of your app instances. You can set up **replicas** to scale the number of instances based on demand.

Kubernetes Deployment Example:

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: myapp:latest
          ports:
            - containerPort: 8080
```

How It Works:

- **Horizontal scaling** helps by creating more instances of your application to handle an increased number of requests.
- **Kubernetes** can automatically scale the number of replicas based on resource usage.

10.2 Load Balancing for Even Traffic Distribution

Once you have multiple instances of your app running, you'll need a **load balancer** to evenly distribute traffic among them. This ensures that no single server becomes overwhelmed, and the requests are distributed optimally.

📌 Example: Load Balancing with Nginx

Nginx is a popular reverse proxy and load balancer for distributing HTTP traffic.

Nginx Configuration Example:

nginx

```
http {
    upstream myapp {
        server app1.example.com:8080;
        server app2.example.com:8080;
        server app3.example.com:8080;
    }

    server {
        location / {
            proxy_pass http://myapp;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
}
```

How It Works:

- **Nginx** will balance the traffic between your app instances.
- Requests will be sent to the instances in a round-robin fashion (or based on other strategies like least connections).

10.3 Database Scaling and Optimization

As your app grows, the database might become a bottleneck. There are several techniques for **scaling** and **optimizing** your database to handle more traffic.

📌 Example: Database Sharding

Sharding involves splitting your database into multiple smaller databases (shards), where each shard stores a subset of your data. This can improve read/write performance and distribute the load more evenly.

Sharding Example:

- For example, if you have user data, you can create multiple shards based on user **ID ranges**:
  - Shard 1: User IDs 1-100,000
  - Shard 2: User IDs 100,001-200,000
  - Shard 3: User IDs 200,001-300,000

How It Works:

- When a user logs in or makes a request, you can route their request to the **correct shard** based on their user ID.
- This approach improves database performance by distributing the load across multiple servers.

📌 Example: Using Read-Replica for Improved Read Performance

You can use **read replicas** to offload read operations from your primary database.

Spring Boot Configuration:

yaml

```
spring:
  datasource:
    url: jdbc:mysql://primary-db-url:3306/mydb
    username: user
    password: password
    driver-class-name: com.mysql.cj.jdbc.Driver
    read-only: false
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect

spring:
  datasource:
    read-only:
      url: jdbc:mysql://read-replica-db-url:3306/mydb
      username: user
      password: password
```

How It Works:

- **Read replicas** can handle read-only queries, reducing the load on the primary database.
- This allows the primary database to focus on **write operations**.

10.4 Caching Strategies for High Traffic

When handling high traffic, **caching** becomes a critical strategy to reduce database queries and improve response times.

📌 Example: Caching with Redis

Redis is an in-memory data store that can be used for caching. It stores frequently accessed data to avoid querying the database repeatedly.

Spring Boot + Redis Configuration:

yaml

```yaml
spring:
  cache:
    type: redis
  redis:
    host: localhost
    port: 6379
    password: mysecret
```

Using Cache Annotations in Spring Boot:

```
@Cacheable("users")
public User getUserById(Long id) {
    return userRepository.findById(id).orElse(null);
}
```

How It Works:

- Redis stores the results of frequently accessed data (like user details).
- The first time the data is requested, it’s fetched from the database and cached in Redis. Subsequent requests are served from the cache, improving response times.

10.5 Monitoring and Automatic Scaling

Monitoring the performance of your application is essential, especially under high traffic conditions. **Cloud platforms** like AWS, Google Cloud, and Azure offer **auto-scaling** and **monitoring tools**.

📌 Example: Using Spring Boot Actuator for Monitoring

As discussed earlier, you can use **Spring Boot Actuator** to monitor your app's health, metrics, and other important information.

Spring Boot Actuator Endpoints:

- /actuator/health: Provides health status of the app.
- /actuator/metrics: Displays various metrics such as JVM memory usage, request counts, etc.

How It Works:

- You can configure auto-scaling based on **CPU usage** or other **performance metrics**.
- If your app is under heavy load, the cloud platform will automatically add more instances.

🎯 Step 10 Summary

✅ **Horizontal scaling** helps your app handle more traffic by deploying it across multiple instances.
✅ **Load balancing** distributes incoming traffic evenly to your app instances.
✅ **Database scaling** with sharding and read replicas improves data access performance.
✅ **Caching with Redis** reduces the load on your database and speeds up response times.
✅ **Monitoring and auto-scaling** ensure that your app can adjust to traffic changes dynamically.