JSF Lifecycle and State Management

A deep dive into how JSF processes requests and maintains state

Overview

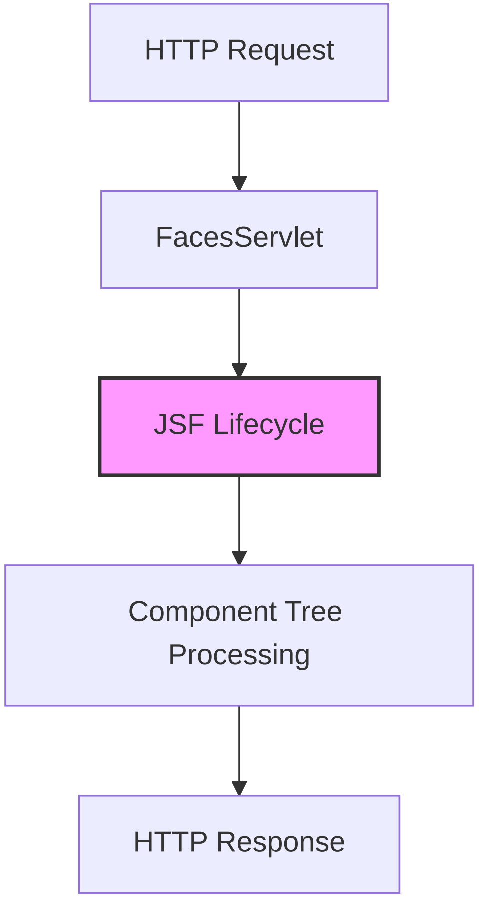
- Understanding the JSF Lifecycle
  - GET Requests vs POST Requests flow
  - The Six Lifecycle Phases
- Controlling the Lifecycle
  - `Immediate` Attribute
  - Phase Listeners
- JSF State Management
  - Why State Management Matters
  - Server-side vs Client-side State Saving
  - Handling `ViewExpiredException`

Introduction to JSF Lifecycle

- JSF follows a **six-phase lifecycle** to process requests and render responses.
- The lifecycle ensures proper handling of user input, validation, model updates, and UI rendering.

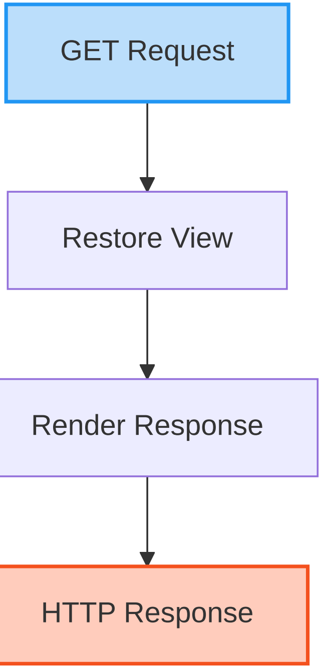
Introduction to JSF Lifecycle

- JSF follows a **six-phase lifecycle** to process requests and render responses.
- The lifecycle ensures proper handling of user input, validation, model updates, and UI rendering.



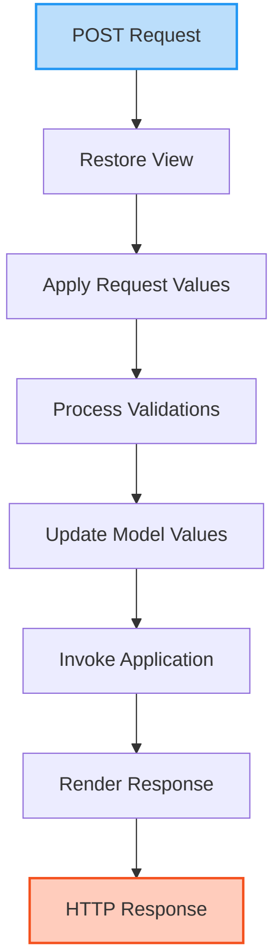
JSF Request Types

GET Request (Initial)



JSF Request Types

POST Request (Postback)



Phase 1: Restore View

- **Initial Request (GET)**:
  - Creates an empty view, since there’s no `UIViewRoot` to restore
  - Advances to Render Response phase
- **Postback Request (POST)**:
  - Restores component tree from ViewState
  - Prepares all components for processing

Phase 2: Apply Request Values

- Extracts values from request parameters
- Stores extracted values in components locally
- Queues ValueChangeEvents

Phase 3: Process Validations

- Converts string values to expected types
- Validates component values
  - JSF built-in validators (required, length, etc.)
  - Bean validation (javax.validation)
  - Custom validators
- If validation fails:
  - Adds error messages to FacesContext
  - Skips to Render Response phase

Phase 3: Process Validations

- Converts string values to expected types
- Validates component values
  - JSF built-in validators (required, length, etc.)
  - Bean validation (javax.validation)
  - Custom validators
- If validation fails:
  - Adds error messages to FacesContext
  - Skips to Render Response phase

```
<h:inputText value="#{user.email}" id="email">
    <f:validateRegex pattern="^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$" />
    <f:ajax event="blur" render="emailError" />
</h:inputText>
<h:message for="email" id="emailError" style="color:red" />
```

Phase 4: Update Model Values

- Component values are set into model properties
  - Updates backing bean or entity properties
  - Type conversion is applied where needed

Phase 4: Update Model Values

- Component values are set into model properties
  - Updates backing bean or entity properties
  - Type conversion is applied where needed

```java
// Inside JSF implementation (simplified)
public void updateModelValues(FacesContext context) {
    for (UIComponent component : getAllInputComponents()) {
        if (component instanceof UIInput) {
            UIInput input = (UIInput) component;
            ValueExpression ve = input.getValueExpression("value");
            if (ve != null) {
                // Set the component's local value into the model
                ve.setValue(context.getELContext(), input.getLocalValue());
                input.setValue(null); // Clear local value
            }
        }
    }
}
```

Phase 5: Invoke Application

- Executes action methods and listeners
- Processes navigation cases depending on the return type of action method
  - `return String` → navigate to new page by creating new request (ViewScoped bean recreated)
  - `return void` → stay on same page and proceeds to next phase (ViewScoped bean persists)
  - `return empty String` → refresh current page (ViewScoped bean recreated)
- Queued events are processed (button clicks, value change events, etc.)

Phase 5: Invoke Application

- Executes action methods and listeners
- Processes navigation cases depending on the return type of action method
  - `return String` → navigate to new page by creating new request (ViewScoped bean recreated)
  - `return void` → stay on same page and proceeds to next phase (ViewScoped bean persists)
  - `return empty String` → refresh current page (ViewScoped bean recreated)
- Queued events are processed (button clicks, value change events, etc.)

```
public String login() {
    if (userService.authenticate(username, password)) {
        return "dashboard?faces-redirect=true"; // Navigation
    }
    FacesContext.getCurrentInstance().addMessage(null,
        new FacesMessage(FacesMessage.SEVERITY_ERROR, "Invalid credentials", null));
    return ""; // Stay on same page
}
```

Phase 6: Render Response

- Renders component tree into HTML
- Saves the state of components for future request processing
- Completes the response

Phase 6: Render Response

- Renders component tree into HTML
- Saves the state of components for future request processing
- Completes the response

```
<!-- Rendered output will include ViewState -->
<form id="j_id1" name="j_id1" method="post" action="/app/login.xhtml" enctype="application/x-www-form-urlencoded">
    <!-- Form components -->
    <!-- Hidden field containing the view state -->
    <input type="hidden" name="jakarta.faces.ViewState" id="j_id1:jakarta.faces.ViewState:0"
           value="-5595324239867351894:6754026291940545952" autocomplete="off" />
</form>
```

The `immediate` Attribute

Alters normal lifecycle processing:

For Input Components:

- Validates during **Apply Request Values** instead of Process Validations
- Useful when you want to prioritize validation for specific inputs
- Example: Username field in a large registration form

The `immediate` Attribute

Alters normal lifecycle processing:

For Input Components:

- Validates during **Apply Request Values** instead of Process Validations
- Useful when you want to prioritize validation for specific inputs
- Example: Username field in a large registration form

For Command Components:

- Action methods execute in **Apply Request Values**
- **Process validations**, **Update model values**, and **Invoke Application** phases are skipped
- Skips conversion and validation of non-immediate input components
- Useful for "Cancel" or "Back" buttons in the form
- Get the non-converted and non-validated input value using `component.getSubmittedValue()`

Quiz: Using `immediate` Attribute

You need to reuse a login form for both login and "forgot password" functionality:

- The "Login" button should validate both username and password
- The "Forgot Password" button should only validate the username

How would you implement this?

Quiz: Using `immediate` Attribute

You need to reuse a login form for both login and "forgot password" functionality:

- The "Login" button should validate both username and password
- The "Forgot Password" button should only validate the username

How would you implement this?

Solution

```
<h:inputText id="username" value="#{loginBean.username}" immediate="true" required="true" />
<h:inputSecret id="password" value="#{loginBean.password}" required="true" />

<h:commandButton value="Login" action="#{loginBean.login}" />
<h:commandButton value="Forgot Password" action="#{loginBean.forgotPassword}" immediate="true" />
```

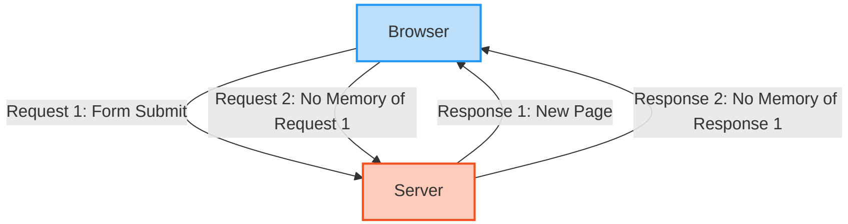
- Username validated early (immediate="true")
- Forgot Password button with immediate="true" skips password validation
- Login button validates both fields

JSF State Management

Stateful JSF

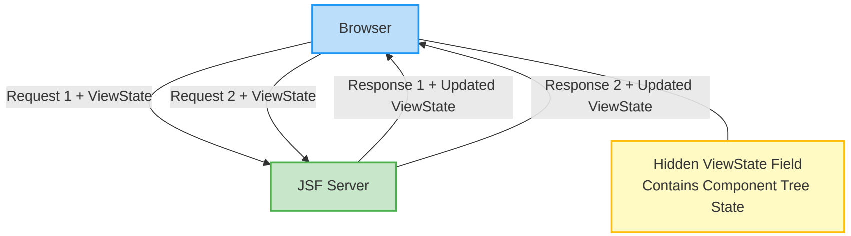
HTTP is Stateless

- Each request is completely independent
- No component state preservation


Browser
Request 1: Form Submit
Request 2: No Memory of Request 1
Response 1: New Page
Response 2: No Memory of Response 1
Server

JSF is Stateful

- Preserves entire UI component tree state
- Maintains form inputs and button states


Browser
Request 1 + ViewState
Request 2 + ViewState
Response 1 + Updated ViewState
Response 2 + Updated ViewState
JSF Server
Hidden ViewState Field Contains Component Tree State

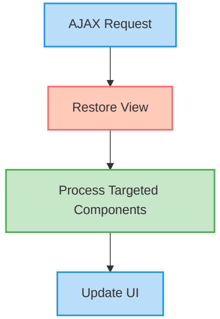
Why JSF Saves State

Form Processing

- **Validate the data**.
- **Update the model** (backing beans).
- **Re-render the form** with the same values and validation message if validation fails.

Dynamic Changes

- The component tree can change during **AJAX updates**.
- JSF does **NOT reload the whole page**.
- Restores the view from `ViewState` and processes only **targeted components**.



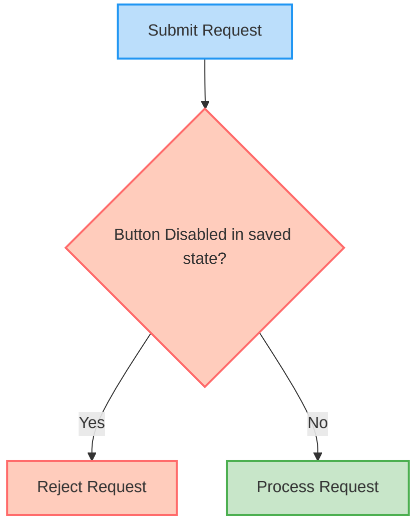
Why JSF Saves State (continued)

Lifecycle Handling

- State provides information on:
  - Request parameters
  - Converters/validators
  - Bound managed bean properties

Security Protection

- Prevents tampered requests (e.g., submitting a disabled button).
- Validates against the saved state during the Apply Request Values Phase.



How JSF’s Saved State Enhances Security

JSF prevents tampered requests through:

How JSF’s Saved State Enhances Security

JSF prevents tampered requests through:

- Re-evaluation of component attributes during lifecycle
  - `rendered`, `disabled`, and `readonly` states are checked
  - Prevents activation of disabled components

How JSF’s Saved State Enhances Security

JSF prevents tampered requests through:

- Re-evaluation of component attributes during lifecycle
  - `rendered`, `disabled`, and `readonly` states are checked
  - Prevents activation of disabled components
- Validation of submitted values
  - `UISelectOne` and `UISelectMany` validate against available options
  - Prevents injection of unauthorized values

How JSF’s Saved State Enhances Security

JSF prevents tampered requests through:

- Re-evaluation of component attributes during lifecycle
  - `rendered`, `disabled`, and `readonly` states are checked
  - Prevents activation of disabled components
- Validation of submitted values
  - `UISelectOne` and `UISelectMany` validate against available options
  - Prevents injection of unauthorized values
- Protection of component tree structure
  - Prevents addition or removal of components via tampered requests
  - Ensures only valid UI interactions are processed

The ViewState

- **Hidden Form Field**: `jakarta.faces.ViewState`
- **Contains**: Serialized component tree state (client-side) or reference ID (server-side)
- **Used During**: Restore View phase to rebuild the component tree

Server-Side vs Client-Side State Saving

Server-Side

- Stores state in session
- Lower bandwidth usage
- Higher server memory usage
- Protected in session
- Risk of ViewExpiredException
- Default in most implementations

```
<!-- web.xml configuration -->
<context-param>
    <param-name>jakarta.faces.STATE_SAVING_METHOD</param-nam
    <param-value>server</param-value>
</context-param>
```

Client-Side

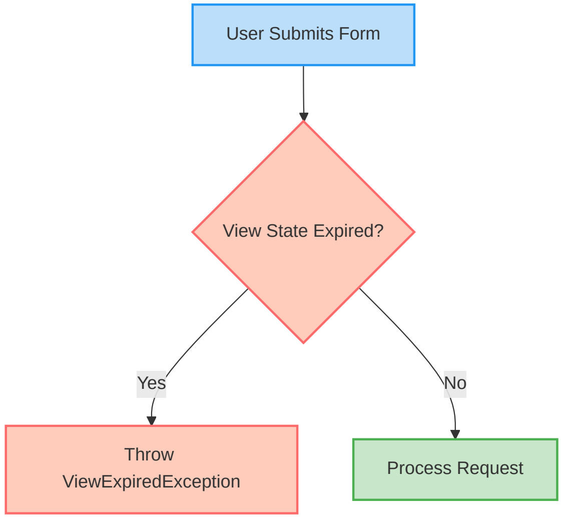
- Serialized state in form's hidden field
- Higher bandwidth usage
- Lower server memory usage
- Encrypted but exposed to client
- Prevents ViewExpiredException
- Better for clustered environments

```
<!-- web.xml configuration -->
<context-param>
    <param-name>jakarta.faces.STATE_SAVING_METHOD</param-nam
    <param-value>client</param-value>
</context-param>
```

ViewExpiredException in JSF

What is `ViewExpiredException`?

- Occurs when JSF cannot restore the **saved state** of a page.
- when server side state (component tree) gets invalidated and user tries to do a postback request on the same page



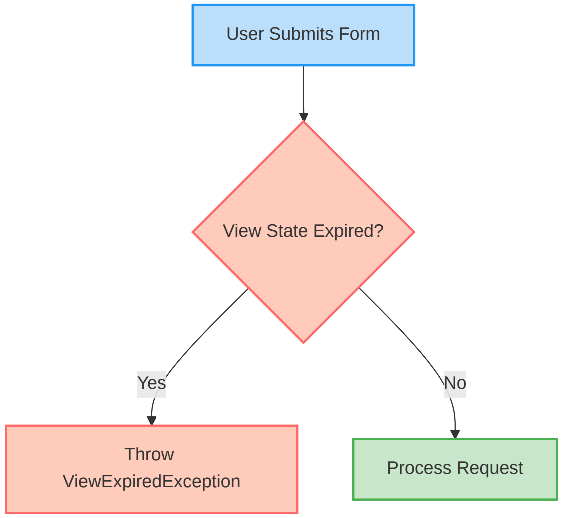
ViewExpiredException in JSF

What is `ViewExpiredException`?

- Occurs when JSF cannot restore the **saved state** of a page.
- when server side state (component tree) gets invalidated and user tries to do a postback request on the same page

Common causes:

- **Session Timeout:** Form left open too long; session expires before submission.
- **Browser Navigation:** User submits, navigates away, then returns and resubmits.
- **Server State Limits:** Server purges old ViewState when limits are exceeded.



ViewExpiredException in JSF (continued)

Prevention & Handling:

- Use client-side state saving for critical forms
- Add session keep-alive for long forms
- Implement proper exception handling

```xml
<error-page>
   <exception-type>jakarta.faces.application.ViewExpiredException</exception-type>
   <location>/viewExpired.xhtml</location>
</error-page>
```

- Use libraries like OmniFaces for better handling

Demo: Inspecting JSF Lifecycle

Phase Listener Implementation

```java
public class LifecycleLogger implements PhaseListener {
    @Override
    public void beforePhase(PhaseEvent event) {
        System.out.println("Before phase: " +
            event.getPhaseId());
    }

    @Override
    public void afterPhase(PhaseEvent event) {
        System.out.println("After phase: " +
            event.getPhaseId());
    }

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
```

faces-config.xml Registration

```xml
<lifecycle>
    <phase-listener>
        com.example.LifecycleLogger
    </phase-listener>
</lifecycle>
```

Debug Output (Initial Request)

```
Before phase: RESTORE_VIEW 1
After phase: RESTORE_VIEW 1
Before phase: RENDER_RESPONSE 6
After phase: RENDER_RESPONSE 6
```

Debug Output (Postback)

```
Before phase: RESTORE_VIEW 1
After phase: RESTORE_VIEW 1
Before phase: APPLY_REQUEST_VALUES 2
After phase: APPLY_REQUEST_VALUES 2
...
```

Thank You!

Any questions?

END