🏗 Architecture Decisions Guide

> **"Why We Built It This Way: The Story Behind Every Design Choice"**

🎯 The Art of Architectural Decision Making

Every line of code tells a story. Every design choice has a reason. This guide documents the architectural decisions made throughout the project, the alternatives considered, and the reasoning behind our choices. Understanding these decisions will help you make better choices in future projects and explain our approach to your team.

📋 Decision Record Template

```
# ADR-XXX: [Decision Title]

## Status
[Proposed | Accepted | Deprecated | Superseded]

## Context
What is the issue that we're seeing that is motivating this decision or change?

## Decision
What is the change that we're proposing or have agreed to implement?

## Consequences
What becomes easier or more difficult to do and any risks introduced by the
change?

## Alternatives Considered
What other options were evaluated and why were they not chosen?
```

🏛 Core Architecture Decisions

ADR-001: Custom Object vs Standard Object Extension

Status: Accepted

Context: We needed to track job applications with specific fields and business logic that don't map well to standard Salesforce objects like Opportunities or Leads.

Decision: Create a custom object `Job_Application__c` with purpose-built fields and relationships.

Alternatives Considered:

Option 1: Extend Opportunity Object

```
// Using Opportunity for job tracking
Opportunity jobOpp = new Opportunity(
    Name = 'Software Engineer - Salesforce',
    StageName = 'Applied',
    CloseDate = Date.today().addDays(30),
    Amount = 120000 // Salary as opportunity amount
);
```

Pros: Leverage existing functionality, familiar to users *Cons*: Forced business model mismatch, unnecessary complexity, confusing reporting

Option 2: Extend Lead Object

```
// Using Lead for job tracking
Lead jobLead = new Lead(
    FirstName = UserInfo.getFirstName(),
    LastName = UserInfo.getLastName(),
    Company = 'Salesforce',
    Status = 'Applied'
);
```

Pros: Natural progression from lead to opportunity *Cons*: Leads are for prospects, not personal job tracking

Our Choice: Custom Object

```
// Clean, purpose-built solution
Job_Application__c app = new Job_Application__c(
    Company_Name__c = 'Salesforce',
    Position_Title__c = 'Software Engineer',
    Status__c = 'Applied',
    Salary__c = 120000,
    Application_Date__c = Date.today()
);
```

Consequences:

- ☑ Perfect fit for business requirements
- ☑ Clean data model and reporting
- ☑ No unnecessary standard object baggage
- ✖ Need to build all functionality from scratch
- ✖ Users need to learn new object

ADR-002: Trigger Framework Architecture

Status: Accepted

Context: Need to implement business logic that fires when job application records are created or updated, specifically task creation based on status changes.

Decision: Implement a simple trigger with handler pattern for maintainability and testability.

Alternatives Considered:

Option 1: Enterprise Trigger Framework (FFLIB)

```
// Complex enterprise framework
public class JobApplicationTriggerHandler extends fflib_SObjectDomain {
    public JobApplicationTriggerHandler(List<Job_Application__c> records) {
        super(records);
    }

    public override void onAfterInsert() {
        // Complex framework overhead
    }
}
```

Pros: Highly scalable, separation of concerns, industry standard *Cons*: Overkill for single object, steep learning curve, added complexity

Option 2: Process Builder/Flow

```
Process Builder: Job Application Status Change
├── Criteria: Status ISCHANGED()
├── Action: Create Task
└── Action: Send Email
```

Pros: No-code solution, visual design, easy for admins *Cons*: Limited complex logic, performance concerns, debugging challenges

Option 3: Direct Trigger Logic

```
trigger JobApplicationTrigger on Job_Application__c (after insert, after update) {
    // All logic directly in trigger - BAD PRACTICE
    for (Job_Application__c app : Trigger.new) {
        if (app.Status__c == 'Interviewing') {
            Task t = new Task(Subject = 'Prepare for interview');
            insert t;
        }
    }
}
```

Pros: Simple, direct approach *Cons*: Not testable, not bulkified, poor maintainability

Our Choice: Simple Handler Pattern

```
trigger JobApplicationTrigger on Job_Application__c (after insert, after update) {
    if (Trigger.isAfter) {
        if (Trigger.isInsert) {
            JobApplicationTriggerHandler.handleAfterInsert(Trigger.new);
        }
        if (Trigger.isUpdate) {
            JobApplicationTriggerHandler.handleAfterUpdate(Trigger.new,
Trigger.oldMap);
        }
    }
}

public class JobApplicationTriggerHandler {
    public static void handleAfterUpdate(List<Job_Application__c> newApps, Map<Id,
Job_Application__c> oldMap) {
        // Clean, testable, bulkified logic
    }
}
```

Consequences:

- ☑ Clean separation of concerns
- ☑ Easily testable
- ☑ Bulkified by design
- ☑ Simple enough for team to understand
- ✗ Not as scalable as enterprise frameworks
- ✗ Manual pattern enforcement

ADR-003: LWC vs Aura Components

Status: Accepted

Context: Need to build interactive user interfaces for salary calculation and interview scheduling with real-time updates and modern user experience.

Decision: Use Lightning Web Components (LWC) for all new component development.

Alternatives Considered:

Option 1: Aura Components

```
// Aura component approach
({
    handleSalaryChange: function(component, event, helper) {
        var salary = event.getParam("value");
        var takeHome = helper.calculateTakeHome(salary);
        component.set("v.takeHomePay", takeHome);
    }
})
```

Pros: Mature ecosystem, lots of documentation, team familiarity *Cons*: Legacy technology, performance limitations, verbose syntax

Option 2: Visualforce + JavaScript

```
<!-- Visualforce approach -->
<apex:page controller="SalaryController">
    <script>
        function calculateSalary() {
            // jQuery-based interactions
        }
    </script>
</apex:page>
```

Pros: Full control over rendering, server-side processing *Cons*: Not mobile-responsive, outdated UX patterns, poor performance

Our Choice: Lightning Web Components

```
// Modern LWC approach
export default class SalaryCalculator extends LightningElement {
    @track salary = 0;
    @track takeHomePay = 0;

    handleSalaryChange(event) {
        this.salary = event.target.value;
        this.calculateTakeHome();
    }

    calculateTakeHome() {
        // Modern JavaScript, reactive updates
        this.takeHomePay = this.salary * 0.75; // Simplified
    }
}
```

Consequences:

- ☑ Modern web standards (ES6+, Web Components)
- ☑ Better performance than Aura
- ☑ Future-proof technology choice
- ☑ Smaller bundle sizes
- ✖ Newer technology with fewer examples
- ✖ Team learning curve

ADR-004: Client-Side vs Server-Side Calculations

Status: Accepted

Context: The salary calculator needs to provide real-time feedback as users type, but also needs to handle complex tax calculations accurately.

Decision: Implement a hybrid approach: simple calculations client-side for responsiveness, complex calculations server-side for accuracy.

Alternatives Considered:

Option 1: All Server-Side

```
// Every change triggers server call
handleSalaryChange(event) {
    this.salary = event.target.value;
    calculateTakeHomePay({ salary: this.salary })
        .then(result => {
            this.takeHomePay = result;
        });
}
```

Pros: Centralized business logic, always accurate *Cons*: Poor user experience, unnecessary server load, network dependency

Option 2: All Client-Side

```
// Complex tax logic in JavaScript
calculateTakeHome() {
    const federalTax = this.calculateFederalTax(this.salary);
    const stateTax = this.calculateStateTax(this.salary, this.state);
    const socialSecurity = this.calculateSocialSecurity(this.salary);
    // ... complex logic duplicated
}
```

Pros: Immediate feedback, no server dependency *Cons*: Logic duplication, maintenance nightmare, potential inconsistencies

Our Choice: Hybrid Approach

```
// Client-side for immediate feedback
calculateTakeHome() {
    // Simple estimation for immediate feedback
    const estimatedTax = this.salary * 0.25;
    this.takeHomePay = this.salary - estimatedTax;
}
```

```
// Server-side for accuracy when needed
async getAccurateCalculation() {
    const result = await calculateAccurateTakeHome({
        salary: this.salary,
        state: this.state
    });
    this.takeHomePay = result.takeHomePay;
    this.breakdown = result.breakdown;
}
```

Consequences:

- ✅ Responsive user experience
- ✅ Accurate calculations when needed
- ✅ Single source of truth for complex logic
- ❌ Slight complexity in managing two calculation paths
- ❌ Potential confusion between estimated and accurate values

---

ADR-005: API Integration Architecture

Status: Accepted

Context: Need to integrate with external APIs for job board data and salary information while handling failures gracefully and maintaining performance.

Decision: Implement a service layer pattern with comprehensive error handling and caching.

Alternatives Considered:

Option 1: Direct API Calls from Components

```
// LWC directly calling APIs - BAD
export default class JobSearch extends LightningElement {
    async searchJobs() {
        // Direct HTTP call from component
        const response = await fetch('/api/jobs');
        this.jobs = await response.json();
    }
}
```

Pros: Simple, direct approach *Cons*: No error handling, no caching, security issues, code duplication

Option 2: Platform Events for Async Processing

```
// Publish event for async processing
JobSearchEvent__e event = new JobSearchEvent__e(
    SearchTerm__c = 'Software Engineer',
    Location__c = 'San Francisco'
);
EventBus.publish(event);
```

Pros: Decoupled, scalable, handles high volume *Cons*: Added complexity, eventual consistency, harder debugging

Our Choice: Service Layer with Error Handling

```
public class ExternalAPIService {

    @AuraEnabled(cacheable=true)
    public static APIResponse searchJobs(String searchTerm) {
        try {
            // Check cache first
            String cacheKey = 'jobs_' + searchTerm;
            Object cachedResult = Cache.Org.get(cacheKey);
            if (cachedResult != null) {
                return (APIResponse) cachedResult;
            }

            // Make API call
            HttpRequest req = buildJobSearchRequest(searchTerm);
            Http http = new Http();
            HttpResponse res = http.send(req);

            if (res.getStatusCode() == 200) {
                APIResponse result = parseJobResponse(res.getBody());
                Cache.Org.put(cacheKey, result, 3600); // 1 hour cache
                return result;
            } else {
                return handleAPIError(res);
            }

        } catch (Exception e) {
            return new APIResponse(false, 'Service temporarily unavailable');
        }
    }
}
```

Consequences:

- ✅ Centralized error handling
- ✅ Caching for performance
- ✅ Consistent API interface
- ✅ Testable and maintainable
- ❌ More complex than direct calls
- ❌ Additional abstraction layer

ADR-006: Testing Strategy

Status: Accepted

Context: Need comprehensive testing to ensure code quality, prevent regressions, and enable confident deployments.

Decision: Implement a multi-layered testing strategy with unit tests, integration tests, and end-to-end validation.

Our Testing Pyramid:

```
     /\     E2E Tests (Few)
    /  \    ├── Manual testing
   /____\   └── Automated UI tests
  /      \
 /________\  Integration Tests (Some)
            ├── API integration tests
            └── Component integration tests

Unit Tests (Many)
├── Apex class tests
├── Trigger handler tests
└── Utility method tests
```

Testing Standards:

```
// Example of comprehensive test
@isTest
public class JobApplicationServiceTest {

    @TestSetup
    static void setupData() {
        // Arrange: Create test data
        TestDataFactory.createJobApplications(10, 'Applied');
    }

    @isTest
    static void testCreateJobApplication_Success() {
        // Arrange
        Job_Application__c testApp =
TestDataFactory.createJobApplication('Interviewing');

        // Act
        Test.startTest();
        Id createdId = JobApplicationService.createJobApplication(testApp);
        Test.stopTest();

        // Assert
        Job_Application__c created = [SELECT Status__c FROM Job_Application__c
WHERE Id = :createdId];
        System.assertEquals('Interviewing', created.Status__c);
        // Verify side effects (tasks created)
        List<Task> tasks = [SELECT Subject FROM Task WHERE WhatId = :createdId];
        System.assertEquals(2, tasks.size(), 'Should create interview preparation
tasks');
    }
}
```

Consequences:

- ✅ High confidence in deployments
- ✅ Regression prevention
- ✅ Documentation through tests
- ✅ Enables refactoring
- ❌ Initial time investment
- ❌ Maintenance overhead

🎯 Key Architectural Principles

🏗 Design Principles We Follow

1. **SOLID Principles**: Single responsibility, open/closed, Liskov substitution, interface segregation, dependency inversion

2. **DRY (Don't Repeat Yourself)**: Shared utilities and common patterns

3. **KISS (Keep It Simple, Stupid)**: Simple solutions over complex ones

4. **YAGNI (You Aren't Gonna Need It)**: Build what you need now, not what you might need

5. **Separation of Concerns**: Clear boundaries between layers

🔧 Salesforce-Specific Principles

1. **Governor Limit Awareness**: Design for scale from day one

2. **Bulkification**: Handle one record or thousands with same code

3. **Security First**: CRUD/FLS checking in all operations

4. **Declarative First**: Use clicks before code when appropriate

5. **User Experience**: Mobile-first, accessible design

📊 Decision Impact Analysis

✅ Decisions That Worked Well

1. **Custom Object Choice**: Perfect fit for requirements
2. **LWC Technology**: Future-proof and performant
3. **Service Layer Pattern**: Clean, testable, maintainable

4. **Comprehensive Testing**: High confidence in changes

🤔 Decisions We Might Reconsider

1. **Simple Trigger Framework**: Might need enterprise framework as we scale
2. **Client-Side Calculations**: Could be confusing with two calculation paths
3. **Synchronous API Calls**: Might need async processing for high volume

📈 Lessons Learned

1. **Start Simple**: Can always add complexity later
2. **Document Decisions**: Future team members need context
3. **Consider Alternatives**: Always evaluate multiple options
4. **Think Long-Term**: Consider maintenance and scalability
5. **Get Feedback**: Team input improves decisions

🚀 Future Architecture Considerations

🔮 Potential Enhancements

1. **Microservices Architecture**: Break into smaller, focused services
2. **Event-Driven Architecture**: Use Platform Events for decoupling
3. **AI/ML Integration**: Einstein features for job recommendations
4. **Mobile App**: Native mobile application
5. **Multi-Org Strategy**: Support for multiple Salesforce orgs

📋 Technical Debt Management

1. **Regular Architecture Reviews**: Quarterly assessment of decisions
2. **Refactoring Sprints**: Dedicated time for technical improvements
3. **Performance Monitoring**: Track and optimize bottlenecks
4. **Security Audits**: Regular security assessments
5. **Technology Updates**: Stay current with platform evolution

---

Remember: Architecture is about making trade-offs. There's rarely a perfect solution, only solutions that are appropriate for the current context. Document your decisions, learn from them, and be prepared to evolve as requirements change! 🌟