



Semester Long Internship Spring 2026

On

Drupal Web Development

Submitted by

Aniruddh S B

School of Computing
SASTRA Deemed University

July 31, 2026

Acknowledgment

I express my sincere gratitude to the FOSSEE project at IIT Bombay for the opportunity to be part of its internship programme. The scale and openness of the FOSSEE ecosystem made it possible for a student contributor to work directly on applications that are used by colleges and universities across the country, and that experience has shaped the way I think about software that has real users.

I am particularly grateful to the FOSSEE web development team for their guidance, technical direction and encouragement throughout this internship. Their reviews shaped the work at every stage: the decision to stop patching individual forms and instead introduce shared services for mail delivery and AJAX handling emerged from these discussions, and it changed the character of the project from a mechanical version upgrade into an exercise in software design. I thank the team members who took the time to read my code carefully, question the choices behind it and point out what could be done better.

This internship, carried out over the period February 2026 – July 2026, has been an enriching learning experience. Working on a mature Drupal codebase that has been maintained across several major versions gave me exposure to problems that do not appear in academic projects: deprecated APIs that must be replaced without altering behaviour, legacy workflows whose intent must be reconstructed from the code itself, and the discipline required to make incremental, reviewable changes to a system that other people depend on.

Finally, I thank everyone at FOSSEE who provided coordination, infrastructure support and timely responses at the various stages of this work.

Aniruddh S B
SASTRA Deemed University

Abstract

The FOSSEE project at IIT Bombay operates a family of Drupal-based web applications that support the FreeCAD and DWSIM initiatives. These applications handle the complete lifecycle of student and faculty contributions: proposal submission, administrative review and approval, code submission, bulk approval of submitted work, and final publication of completed labs, flowsheets, custom models and textbook companion entries. The custom modules behind these workflows have been carried forward across successive major versions of Drupal, and in the process they accumulated code written against APIs that no longer exist in current releases, along with a substantial amount of duplicated logic.

This report documents a semester-long internship whose objective was to migrate these custom modules from Drupal 10 to Drupal 11 and, in the same pass, to modernize their internal structure. The work covered four custom modules — FreeCAD Lab Migration, DWSIM Custom Model, DWSIM Flowsheet and DWSIM Textbook Companion — comprising roughly sixty form classes and one hundred and forty routes in total.

The migration itself involved a systematic audit of deprecated and removed APIs, followed by their replacement with supported equivalents: procedural mail calls were replaced by the mail plugin manager, path-argument helpers by route parameters, static service lookups by constructor injection, and the version constraints in every module were widened to admit Drupal 11. Beyond the mechanical upgrade, two reusable services were introduced. A centralized mail service replaced per-module mail logic that had been copied between forms, and an AJAX helper service replaced callback methods that were repeated verbatim across multiple form classes. An Export CSV feature was also designed and implemented, allowing administrators to export proposal data filtered by approval status and year of submission.

The result is a codebase that runs on Drupal 11, contains materially less duplicated logic, routes all outbound notification mail through a single tested path, and exposes a reporting capability that the administrative team previously did not have. The report describes not only what was changed but why each design decision was taken, since the reasoning behind the refactoring is the part most likely to be useful to whoever maintains these modules next.

Keywords: Drupal 11, legacy migration, deprecated API replacement, dependency injection, service container, code deduplication, AJAX, mail service, CSV export, FOSSEE, open-source development.

Contents

Acknowledgment	i
Abstract	ii
1 Introduction	1
1.1 Background	1
1.2 The FOSSEE Project	1
1.3 The Application Landscape	2
1.4 Motivation for the Drupal 11 Migration	3
1.5 Objectives of the Internship	5
1.6 Scope and Limitations	5
1.7 Organisation of the Report	6
2 Technical Background and Literature Survey	7
2.1 Drupal as an Application Framework	7
2.2 The Service Container and Dependency Injection	8
2.3 The Form API	9
2.4 The Mail Subsystem	9
2.5 Drupal's Deprecation Policy	10
2.6 What Changed Between Drupal 10 and Drupal 11	10
2.7 Supporting Tools	11
3 Problem Statement and Approach	13
3.1 Problem Statement	13
3.2 Parts of the Problem	13
3.3 Approach	14
3.3.1 Why an audit-first strategy	14
3.3.2 Order of work	15
3.3.3 Incremental delivery	16

3.4	Success Criteria	16
4	Development Environment and Workflow	17
4.1	Overview	17
4.2	Technologies Used	18
4.3	Containerised Setup	18
4.4	Composer and the Core Upgrade	19
4.5	Module Metadata	20
4.6	Git Workflow	20
4.7	Daily Cycle	21
5	Drupal 11 Migration and Deprecated API Review	22
5.1	Migration Pipeline	22
5.2	Inventory of Deprecated and Removed APIs	22
5.3	Mail API Migration	23
5.4	Path Argument Migration	24
5.5	Messenger and Link Generation	25
5.6	Database Access	25
5.7	Dependency Injection	26
5.8	Removal of Obsolete Code	27
5.9	Outcome	27
6	Reusable AJAX Architecture	29
6.1	The Pattern Being Implemented	29
6.2	The Architecture as Inherited	29
6.2.1	Duplicated callback methods	29
6.2.2	Duplicated option builders	30
6.2.3	Why this mattered	31
6.3	The AjaxHelper Service	31
6.3.1	Design decisions	32
6.3.2	Implementation	32
6.3.3	Use from a form	33
6.4	Results	34
7	Centralized Mail Service	35
7.1	Why Mail Matters Here	35
7.2	The Situation as Inherited	35

7.2.1	A removed API, called 65 times	35
7.2.2	Message composition scattered across forms	36
7.2.3	The cross-module requirement	36
7.3	The MailService Design	37
7.3.1	Responsibilities	37
7.3.2	Implementation	37
7.3.3	Registration	38
7.3.4	The resulting call site	39
7.4	Cross-Module Integration	39
7.5	Results	40
8	Export CSV Implementation	41
8.1	Requirement	41
8.2	Design Decisions	41
8.2.1	A route on the existing administrative screen	41
8.2.2	A form, not a link	42
8.2.3	Filter values derived from the data	42
8.2.4	Both filters default to Any	43
8.3	Implementation	43
8.3.1	Routing	43
8.3.2	The form	44
8.3.3	Query construction	45
8.3.4	Response generation	45
8.4	Delivered Interface	46
8.5	Reuse Across Modules	47
9	Module-wise Implementation	48
9.1	FreeCAD Lab Migration	48
9.1.1	Purpose	48
9.1.2	Architecture	49
9.1.3	Migration work	49
9.1.4	Challenges	49
9.1.5	Results	50
9.2	DWSIM Custom Model	51
9.2.1	Purpose	51
9.2.2	Workflow	51

9.2.3	Migration work	51
9.2.4	Challenges	52
9.2.5	Results	52
9.3	DWSIM Flowsheet	52
9.3.1	Purpose	52
9.3.2	Migration work	52
9.3.3	Challenges	53
9.3.4	Results	53
9.4	DWSIM Textbook Companion	53
9.4.1	Purpose	53
9.4.2	Structure	53
9.4.3	Migration work	53
9.4.4	Challenges	54
9.4.5	Results	54
9.5	Consolidated View	55
10	Application Workflow	56
10.1	The Common Workflow	56
10.2	Stage by Stage	57
10.2.1	Proposal creation	57
10.2.2	Administrative review	57
10.2.3	Approval decision	57
10.2.4	Code and file submission	57
10.2.5	Bulk approval	58
10.2.6	Completion	58
10.3	Administrative Workflow	58
10.4	Notification Map	58
11	Testing and Validation	60
11.1	Testing Strategy	60
11.2	Test Matrix	61
11.3	Mail Verification	61
11.4	Regression Testing	62
11.5	Browser and Environment Testing	62
11.6	Cache Rebuild Discipline	62
11.7	Limitations of the Testing Performed	63

12 Challenges Faced	64
12.1 Composer and the Dependency Graph	64
12.2 Container Environment	65
12.3 Working Without Reliable Error Messages	65
12.4 Legacy Code Whose Intent Is Unrecoverable	65
12.5 The Discipline of Scope	66
13 Results	67
13.1 Migration Outcomes	67
13.2 Legacy API Elimination	67
13.3 Reduction in Duplication	68
13.4 Functional Additions	68
13.5 Qualitative Outcomes	68
13.6 Deliverables	69
14 Learning Outcomes	70
14.1 Technical	70
14.2 Software Engineering Practice	71
14.3 Open Source Development	72
15 Conclusion and Future Scope	73
15.1 Conclusion	73
15.2 Future Scope	74
16 Bibliography	76
Bibliography	76
A Repositories and Version Control	78
A.1 Repositories Worked On	78
A.2 Branching Convention	78
A.3 Commit History	79
B Internship Timeline	83

List of Figures

2.1	Layered architecture of the FOSSEE Drupal applications	8
3.1	Iterative approach followed for each module	14
5.1	Five-stage migration pipeline applied to each module	22
6.1	AJAX request cycle for a cascading dropdown	29
6.2	AJAX option-building before and after consolidation	32
7.1	MailService architecture: all dispatch flows through one component . . .	37
8.1	The All Proposals administrative screen, with the Export CSV tab added alongside the existing management tabs	42
8.2	The Export CSV interface with Status and Year of Submission filters, both defaulting to <i>Any</i>	43
8.3	Export CSV with filters applied: approved proposals from 2026, with the matching records shown for confirmation	47
9.1	Completed Labs listing on the migrated FreeCAD site	50
9.2	DWSIM Custom Model workflow	51
10.1	End-to-end workflow common to the FOSSEE Drupal applications	56
A.1	<code>freecad_lab_migration</code> , branch <code>sli-intern</code> : Export CSV with filters and the AJAX table, followed by the module configuration, service logic and form migration work	79
A.2	<code>freecad_lab_migration</code> , branch <code>sli-intern</code> : reusable mail service in- tegration and the refactoring of AJAX callback handling across the lab migration forms	80
A.3	<code>dwsim_custom_model</code> , branch <code>sli-intern</code> : the reusable AJAX and mail helpers introduced for the DWSIM module and registered into the exist- ing module flow, alongside the Drupal 11 module configuration files . . .	80

A.4	<code>dwsim_custom_model</code> , branch <code>sli-intern</code> : controller, form classes, routing and mail service work, with the caching and menu adjustments made for the manage-proposal screens	81
A.5	<code>dwsim_drupal10</code> , branch <code>sli-intern</code> : DWSIM Flowsheet migration — forms, controller and routing moved to Drupal 11 APIs, with the reusable AJAX and mail helpers added and the services registered	81
A.6	<code>dwsim_drupal10</code> , branch <code>sli-intern</code> : lab migration module metadata and service registration, controller and form migration, and the integration of the reusable services into <code>dwsim_custom_model</code>	82
A.7	<code>dwsim_drupal10</code> , branch <code>sli-intern</code> : the final phase — Textbook Companion modules, controller and routing migrated to Drupal 11, custom model compatibility fixes, and the flowsheet abstract approval mail service fix	82

List of Tables

1.1	Custom modules covered by the internship	3
1.2	Size of the custom modules at the start of the internship	3
1.3	Occurrences of legacy and removed APIs found during the initial audit .	4
2.1	Drupal 10 and Drupal 11 compared, on the axes relevant to this work . .	11
4.1	Technologies used during the internship	18
5.1	Deprecated and removed APIs found, and their replacements	23
5.2	Migration summary by module	28
6.1	AJAX implementation surface before consolidation	31
6.2	Effect of AJAX consolidation	34
7.1	Legacy mail call sites found during the audit	35
7.2	Effect of mail centralization	40
8.1	Export CSV: exported fields	47
9.1	FreeCAD Lab Migration: structure	49
9.2	Work performed per module	55
10.1	Workflow transitions that dispatch notifications	59
11.1	Workflow test matrix	61
12.1	Summary of challenges and resolutions	64
13.1	Migration status by module	67
13.2	Legacy API occurrences before and after	67
13.3	Duplication before and after consolidation	68
13.4	Deliverables	69

A.1	Repositories and their role in the internship	78
B.1	Phase-wise summary of the internship	83

Chapter 1

Introduction

1.1 Background

Web applications that support academic programmes tend to have unusually long lifetimes. A portal built to collect student project proposals is rarely retired; it accumulates features, absorbs new categories of content, and continues to serve users year after year while the platform underneath it moves through several major versions. The FOSSEE web applications are of exactly this kind. They were first built on Drupal 6 and Drupal 7, were ported to Drupal 8 when the framework was rewritten on top of Symfony, and have since been carried forward through Drupal 9 and Drupal 10.

Each of those transitions left traces in the code. A function that was idiomatic in Drupal 7 survives as a commented block next to its replacement; a form class that was mechanically converted from a procedural form builder retains the naming conventions of the older API; a query written with the deprecated `db_query()` helper is rewritten using the database service but the surrounding control flow is left untouched. None of this is unusual and none of it is the fault of any individual contributor. It is the normal sediment that collects in a long-lived codebase maintained by a rotating set of student developers, each of whom is asked to make a specific change without breaking anything else.

The consequence is that a version upgrade is never purely mechanical. Drupal’s major releases remove APIs that were formally deprecated in earlier versions, so code that has been dragged forward without being genuinely modernized eventually stops working. Drupal 11, released in 2024, removed a large body of code that had been marked deprecated during the Drupal 10 cycle and raised the minimum requirements for both PHP and Symfony. Any custom module that still calls a removed function will produce a fatal error rather than a warning.

1.2 The FOSSEE Project

FOSSEE — Free/Libre and Open Source Software for Education — is a project funded by the Ministry of Education, Government of India, and hosted at the Indian Institute

of Technology Bombay. Its mandate is to promote the use of open-source software in academia, and it pursues this through several parallel activities: creating documentation and spoken tutorials, running fellowships and internships, organising migration and case-study contests, and developing software tools and the web platforms that support all of the above.

Two of these initiatives are directly relevant to this internship:

- **FreeCAD.** FreeCAD is an open-source parametric 3D modeller. The FOSSEE FreeCAD activity encourages institutions to migrate laboratory experiments from proprietary CAD tools to FreeCAD. Proposals are submitted through a web portal, reviewed by the FOSSEE team, and the resulting models and documentation are published once complete.
- **DWSIM.** DWSIM is an open-source process simulator used in chemical engineering. The FOSSEE DWSIM activity runs several parallel programmes — flowsheeting projects, custom thermodynamic and unit-operation models, and a textbook companion project in which solved examples from standard textbooks are reproduced as DWSIM simulations.

Both initiatives share the same underlying process. A contributor submits a proposal; a FOSSEE reviewer approves or rejects it; the contributor then uploads code, models or simulation files against the approved proposal; a reviewer approves the submitted artefacts, often in bulk; and finally the work is marked complete and published on the public-facing side of the site. Because the process is common, the software that implements it is also structurally similar across modules — a fact that turns out to be central to this report, since it is precisely this similarity that had been expressed as copied code rather than as shared services.

1.3 The Application Landscape

The work described here concerns four custom Drupal modules distributed across two code repositories and two Drupal sites.

Table 1.1: Custom modules covered by the internship

Module	Machine name	Responsibility
FreeCAD Lab Migration	<code>lab_migration</code>	Lab migration proposals, solution proposals, experiment and solution uploads, certificates, completed and in-progress lab listings.
DWSIM Custom Model	<code>custom_model</code>	Custom model proposals, abstract approval, code submission, bulk approval and completion tracking.
DWSIM Flowsheet	<code>dwsim_flowsheet</code>	Flowsheet project proposals, approval workflow and publication.
DWSIM Textbook Companion	<code>textbook_companion</code>	Book proposals, chapter and example management, code review and publication of solved examples.

The FreeCAD Lab Migration module lives in its own repository, named `freecad_lab_migration`. The three DWSIM modules are maintained inside the site repository `dwsim_drupal10`, under `modules/custom/`. A local Drupal 11 site repository was used as the working environment in which the migrated modules were installed and exercised.

Table 1.2 gives the scale of the codebase as it stood at the beginning of the internship. These figures were obtained by counting files and route definitions in the repositories, and they are the numbers that governed how the work was planned: a module with forty-eight routes and sixteen form classes cannot be migrated in a single sitting, and any change that has to be repeated once per form is worth converting into a shared service.

Table 1.2: Size of the custom modules at the start of the internship

Module	Forms	Controllers	Services	Routes
FreeCAD Lab Migration	20	1	2	42
DWSIM Textbook Companion	16	1	2	48
DWSIM Custom Model	12	1	2	27
DWSIM Flowsheet	10	1	2	27
Total	58	4	8	144

1.4 Motivation for the Drupal 11 Migration

The decision to move to Drupal 11 was not driven by a desire for new features. It was driven by three concerns that apply to any production application built on a framework with a fixed release cadence.

Security support

Drupal’s security team supports only the current and, for a limited window, the immediately preceding major version. Once support for a version ends, security advisories are no longer issued for it, and a site running on it must either be upgraded or accepted as a known risk. For applications that accept file uploads from the public and store contributor contact details, remaining on a supported version is not optional.

Dependency ceilings

Drupal 10 pins Symfony 6 and supports PHP 8.1 through 8.3. Drupal 11 moves to Symfony 7 and requires PHP 8.3 or later. Because Composer resolves the dependency graph for the whole site at once, a site that stays on Drupal 10 also stays on the old Symfony branch, and contributed modules that begin requiring Symfony 7 components become uninstallable. Version constraints therefore propagate: the longer the core upgrade is deferred, the more of the surrounding ecosystem becomes unavailable.

Accumulated deprecation debt

This is the concern that dominated the actual work. Drupal deprecates an API in a minor release, keeps it functional for the remainder of the major cycle, and removes it at the next major release. Code that was never updated during the Drupal 10 cycle therefore fails abruptly on Drupal 11. A static audit of the four modules found a large volume of such code, summarized in Table 1.3.

Table 1.3: Occurrences of legacy and removed APIs found during the initial audit

Module	<code>drupal_mail()</code>	<code>language_default()</code>	<code>arg()</code>	Static calls
FreeCAD Lab Migration	11	15	32	689
DWSIM Textbook Companion	44	44	77	74
DWSIM Custom Model	7	4	24	251
DWSIM Flowsheet	3	3	22	315

The counts include occurrences inside commented-out blocks, which were retained in the audit deliberately: commented legacy code is itself a maintenance hazard, because it documents an approach that no longer works and invites a future developer to uncomment it. The `drupal_mail()` figure is the most serious of these. That function belongs to Drupal 7; every call site to it is a latent fatal error and, in practice, a notification that never reaches its recipient.

The second column of the table is equally telling in a different way. The very high count of static service calls — expressions of the form `\Drupal::service(...)` scattered through form and controller classes — is not an error and does not break on Drupal 11. It is, however, the signature of code that has been converted from procedural Drupal 7 style by substitution rather than by design, and it is what makes the code hard to test and hard to reason about.

1.5 Objectives of the Internship

The objectives, as agreed with the mentor at the start of the internship, were:

1. To audit the custom modules for deprecated, removed and legacy API usage, and to produce a prioritized inventory of what had to change.
2. To migrate the FreeCAD Lab Migration module from Drupal 10 to Drupal 11, updating form classes, controllers, routing, services and dependency injection, and removing obsolete code.
3. To eliminate duplicated AJAX callback implementations by designing a reusable AJAX helper service, and to adopt it in the forms that previously carried copied callbacks.
4. To replace the per-module mail logic with a single reusable mail service, and to integrate it across the modules — including the case where one module must invoke the mail routine that lives in another.
5. To design and implement an Export CSV feature allowing administrators to export proposal data, filtered by approval status and year of submission.
6. To apply the same migration and modernization treatment to the DWSIM Custom Model, DWSIM Flowsheet and DWSIM Textbook Companion modules, and to upgrade the site itself to Drupal 11.
7. To maintain the work on a dedicated branch, `sli-intern`, and to push changes for review at regular intervals rather than as a single large merge.

1.6 Scope and Limitations

It is worth being explicit about what this internship did not cover, because the boundary was a deliberate one.

The work was confined to the custom modules and to the site configuration required to run them. Content migration — moving existing proposal, experiment and solution records from the production database into the upgraded site — was performed only to the extent needed for testing, using database dumps in a local environment. The theme layer was not rewritten; the administrative screens continue to use the standard Drupal administration theme, and the public-facing pages continue to use the existing FreeCAD and DWSIM themes. Contributed modules were updated to Drupal 11-compatible releases where such releases existed, but no contributed module was patched or forked.

Automated test coverage was not introduced. This is discussed in the future scope (Chapter 15) as the single most valuable next step, and its absence is the main reason the testing described in Chapter 11 is manual and workflow-driven rather than programmatic.

1.7 Organisation of the Report

The report is organised as follows.

Chapter 2 surveys the technical background: the architecture of Drupal since version 8, the service container and dependency injection, the deprecation policy, and the specific differences between Drupal 10 and 11 that affected this work.

Chapter 3 states the problem formally and describes the approach that was followed, including the audit-first strategy adopted after the initial attempts at direct migration proved unreliable.

Chapter 4 describes the development environment — container setup, Composer, Drush, and the Git workflow used to submit work for review.

Chapter 5 is the core migration chapter. It presents the deprecated API inventory and explains, category by category, what each replacement was and why it was chosen.

Chapters 6 and 7 describe the two reusable services that were introduced: the AJAX helper and the mail service. Each chapter contrasts the old architecture with the new one and quantifies the reduction in duplication.

Chapter 8 documents the Export CSV feature, from requirement to implementation, with screenshots of the delivered interface.

Chapter 9 treats each of the four modules individually — purpose, architecture, migration specifics, challenges and results.

Chapter 10 describes the end-to-end application workflow as it now operates, and Chapter 11 describes how that workflow was validated.

Chapters 12 through 15 cover the difficulties encountered, the consolidated results, the learning outcomes, and the conclusion with future scope. The appendices list the repositories, the branch used, and a week-by-week summary of the internship.

Chapter 2

Technical Background and Literature Survey

This chapter establishes the technical vocabulary used in the rest of the report. It covers the architecture of modern Drupal, the mechanisms that make dependency injection possible, the project’s deprecation policy, and the concrete differences between Drupal 10 and Drupal 11 that had to be accommodated.

2.1 Drupal as an Application Framework

Drupal is often described as a content management system, which understates what it is when used the way FOSSEE uses it. The FOSSEE applications store almost none of their data as Drupal nodes. Proposals, experiments, solutions, books, chapters and flowsheets live in custom database tables declared by the modules themselves through `hook_schema()`. Drupal supplies the HTTP kernel, routing, the form system, the database abstraction layer, the user and permission model, the mail subsystem and the render pipeline; the modules supply the domain.

This distinction matters for a migration. Because the applications bypass the entity system, they are insulated from changes to entity and field APIs, which is where a large part of Drupal’s churn occurs. Conversely, they are heavily exposed to changes in the form, routing and service APIs, which is exactly where the work concentrated.

Since Drupal 8 the framework has been built on Symfony components. The request lifecycle is the Symfony `HttpKernel` lifecycle: a request enters, the router resolves it to a controller, the controller returns a response or a render array, and the render pipeline turns the latter into HTML. Figure 2.1 shows the layering as it applies to these modules.

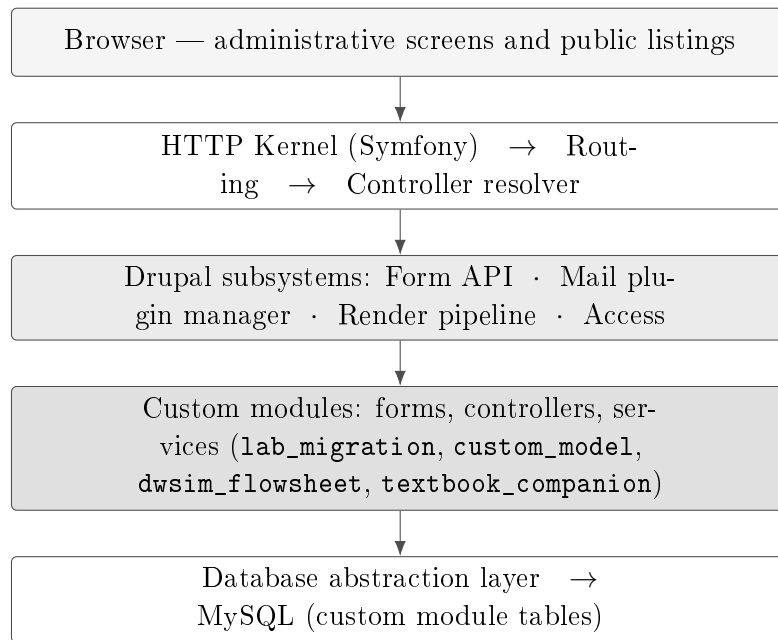


Figure 2.1: Layered architecture of the FOSSEE Drupal applications

2.2 The Service Container and Dependency Injection

Drupal maintains a service container: a registry of shared objects, each declared in a `*.services.yml` file with a class and an ordered list of constructor arguments. When a service is first requested, the container instantiates it, resolving each argument — which may itself be another service — and caches the result.

There are two ways for code to obtain a service, and the difference between them is the single most important architectural theme of this report.

The first is the static accessor, `\Drupal::service('name')`. It works anywhere and requires no ceremony, which is why it appears hundreds of times in the modules as they were inherited. Its cost is hidden: a class that fetches its collaborators from a global registry does not declare what it depends on. Reading the class tells you nothing about what it needs; you have to read every method body. It also cannot be unit-tested without a bootstrapped container, because there is no way to substitute a test double for something the class fetches on its own.

The second is constructor injection. The class declares its dependencies as constructor parameters, stores them as properties, and the container supplies them — either through a `create()` factory method for forms and controllers, or through the `arguments` key in the service definition for plain services. This is more verbose at the point of declaration and considerably clearer everywhere else.

```

1 // Service location: dependency is invisible from outside the method.
2 public function submitForm(array &$form, FormStateInterface $form_state) {
3     $mailManager = \Drupal::service('plugin.manager.mail');
4     $mailManager->mail('lab_migration', 'proposal_received', $to, 'en', $params);
5 }
6
7 // Constructor injection: dependency is declared once, visible, replaceable.

```

```

8 public function __construct(MailManagerInterface $mail_manager) {
9     $this->mailManager = $mail_manager;
10 }
11
12 public static function create(ContainerInterface $container) {
13     return new static($container->get('plugin.manager.mail'));
14 }

```

Listing 2.1: Service location versus constructor injection

Both forms are valid on Drupal 11. Neither one breaks the migration. The reason injection was adopted anyway is that the modules were about to acquire two new shared services — the mail service and the AJAX helper — and adding those through static lookups would have reproduced the very pattern that made the existing code difficult to work with.

2.3 The Form API

A Drupal form is a class implementing `FormInterface`, usually by extending `FormBase`. It declares an ID, builds a render array in `buildForm()`, validates in `validateForm()` and acts in `submitForm()`. Elements in the render array may carry an `'#ajax'` key, which registers a callback invoked when the element changes; the callback returns a fragment of the form that replaces a designated wrapper element in the page.

The cascading-dropdown pattern built on this mechanism appears throughout the FOSSEE modules: choosing a lab populates the experiment dropdown, choosing an experiment populates the solution dropdown, choosing a solution reveals the file list. Chapter 6 examines how this pattern was implemented before the internship and how it was consolidated.

2.4 The Mail Subsystem

Drupal’s mail subsystem separates message composition from delivery. A module implements `hook_mail()`, which is passed a message key and a parameter array and is expected to populate the subject and body. Delivery is handled by a mail plugin obtained from the `plugin.manager.mail` service, which chooses an implementation based on configuration — PHP’s `mail()` by default, SMTP or a transactional provider if a contributed module supplies one.

In Drupal 7 the entry point was the procedural function `drupal_mail($module, $key, $to, $language, $params, $from, $send)`. That function no longer exists. The supported equivalent is:

```

1 // Removed API (Drupal 7).
2 drupal_mail('lab_migration', 'proposal_received', $email_to,
3     language_default(), $param, $from, TRUE);
4
5 // Supported API (Drupal 8 onwards).
6 $this->mailManager->mail(
7     'lab_migration', // module
8     'proposal_received', // key handled by hook_mail()

```

```

9  $email_to, // recipient
10 $this->languageManager->getDefaultLanguage()->getId(),
11 $params, // data for hook_mail()
12 $from,
13 TRUE // send immediately
14 );

```

Listing 2.2: Drupal 7 mail call and its modern equivalent

The signature is similar enough that the substitution looks trivial, and this similarity is a trap. The parameter that changed meaning is the language: the old function accepted a language object, whereas the plugin manager expects a langcode string. Passing the object silently produces a message whose translation context is wrong.

2.5 Drupal’s Deprecation Policy

Drupal follows a predictable lifecycle. An API is marked deprecated in a minor release, annotated with the version in which it will be removed, and continues to function for the remainder of the major cycle while emitting a deprecation notice. It is removed in the next major release.

The practical consequence for a maintainer is that upgrade work should be done continuously during a major cycle, not at its end. Deprecation notices are visible in the logs of a Drupal 10 site for exactly the code that will break on Drupal 11 — but only if someone reads them. When the work is deferred, the notices become fatal errors all at once, which is the situation these modules were in.

2.6 What Changed Between Drupal 10 and Drupal 11

Table 2.1 summarizes the differences relevant to this project.

Table 2.1: Drupal 10 and Drupal 11 compared, on the axes relevant to this work

Aspect	Drupal 10	Drupal 11
PHP requirement	8.1 minimum	8.3 minimum
Symfony branch	Symfony 6	Symfony 7
Deprecated code	Drupal 9 deprecations removed; Drupal 10 deprecations present and functional	Drupal 10 deprecations removed
JavaScript	jQuery present; once available as a standalone library	Continued reduction of jQuery dependence in core
Testing	PHPUnit 9	PHPUnit 10
CKEditor	CKEditor 5	CKEditor 5
Module metadata	<code>core_version_requirement</code> admits <code>~10</code>	admits <code>~11</code>

The last row is deceptively small. A module whose `info.yml` declares the constraint

```
core_version_requirement: ~8 || ~9 || ~10
```

is simply invisible to a Drupal 11 site — it cannot be installed and does not appear in the module list. All four modules carried exactly this constraint, so widening it was the precondition for doing anything else.

2.7 Supporting Tools

Composer

Composer manages PHP dependencies and, since Drupal 8, is the supported way to manage Drupal itself. The site is described by `composer.json`, and the exact resolved versions are recorded in `composer.lock`. A core upgrade is performed by changing the constraint on `drupal/core-recommended` and allowing Composer to resolve the graph. When it cannot, it reports the conflict — usually a contributed module whose constraint excludes the new core version.

Drush

Drush is the command-line interface to a Drupal site. The commands used constantly during this work were `drush cr` (rebuild caches), `drush updb` (run database updates), `drush en` and `drush pmu` (enable and uninstall modules), `drush ws` (tail the watchdog log) and `drush sqlc` (open a database shell).

The importance of `drush cr` is difficult to overstate for anyone new to Drupal. Routing, services, plugin definitions and hook implementations are all cached. A newly added route, a changed service definition or a new `hook_mail()` implementation does not take effect until the cache is rebuilt, and the symptom of forgetting is a page-not-found or a service-not-found error that looks like a code bug and is not one.

Upgrade Status and Drupal Rector

The Upgrade Status contributed module scans an installed site's custom and contributed code for calls to deprecated APIs and reports them per module, distinguishing issues that must be fixed before the next major version from those that are merely advisory. Drupal Rector automates a subset of the resulting changes using rule-based refactoring.

Both tools were used during the audit phase. Their limitation, discussed further in Chapter 5, is that they detect what they have rules for. They reliably find calls to functions deprecated through the standard annotation mechanism, but the FOSSEE modules also contain code carried over from Drupal 7 to calls that were removed several major versions ago and are therefore no longer in any deprecation database. Those had to be found by direct search of the source.

Containerisation

The development environment was containerised, using Docker on one machine and Podman on another. The motivation was the PHP version requirement: Drupal 11 requires PHP 8.3, and installing that system-wide alongside an existing PHP installation is disruptive. A container fixes the PHP version, the web server and the database version as part of the environment definition, so the same stack can be reproduced on a different machine without touching the host.

Chapter 3

Problem Statement and Approach

3.1 Problem Statement

To migrate the custom Drupal modules used by the FOSSEE FreeCAD and DWSIM activities from Drupal 10 to Drupal 11, and to improve their internal structure along the way, without changing how the applications behave for their users.

In practice this meant four things: replacing the deprecated APIs that Drupal 11 no longer supports, removing the duplicated AJAX and mail code by moving it into reusable services, adding a CSV export facility for administrators, and keeping every existing workflow working exactly as before.

The last of these is the important one. These applications have live users and real data. A migration that produces cleaner code but changes the order in which a proposal moves from submission to completion, or changes who receives which notification, has not succeeded.

3.2 Parts of the Problem

The work divided into four parts, each needing a different approach.

Compatibility

Code that calls removed APIs has to be found and rewritten. Finding it is a search problem; rewriting it is mostly mechanical. It is not fully automatic, though, because some call sites need a judgement rather than a substitution. Replacing `arg(3)` with a route parameter, for example, requires knowing which route the code is reached through.

Duplication

The same logic appears in several places, so fixing a bug means finding every copy, and any copy that is missed becomes a divergence. The solution is to identify the shared

responsibility, move it into a service, and have the original call sites use that service.

Missing functionality

The administrative team had no way to extract proposal data for reporting — the information was visible on screen and nowhere else. This was ordinary feature work, and it also served as a useful test of the migrated architecture.

Verification

Every change had to be shown not to break an existing workflow, and there was no automated test suite to rely on. This was handled by running each workflow from start to finish after every significant change (Chapter 11).

3.3 Approach

The approach that was eventually adopted is shown in Figure 3.1. It is worth noting that this was not the approach taken at the outset.

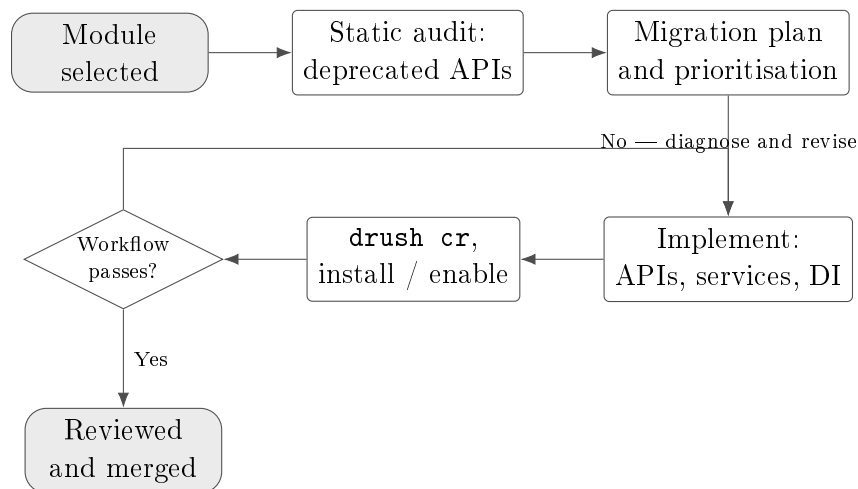


Figure 3.1: Iterative approach followed for each module

3.3.1 Why an audit-first strategy

The first attempt at migrating the FreeCAD Lab Migration module followed the obvious path: bump the core version constraint, install the module on a Drupal 11 site, and fix whatever broke. This proved unproductive for a reason that is specific to how Drupal fails.

A fatal error in a Drupal module does not usually produce a helpful message in the browser. It produces a white screen or a generic error page, with the actual exception recorded in the watchdog log or the PHP error log. Worse, many of the failure sites

in this codebase are inside mail-sending branches or administrative approval paths that are only reached after several steps of a workflow. Finding a broken `drupal_mail()` call by triggering it means submitting a proposal, logging in as an administrator, navigating to the approval screen, approving, and only then discovering the failure — and then repeating the entire sequence after each fix.

Auditing the source first inverts this. A search for known-removed function names across the module produces the complete list of failure sites in seconds, whether or not they are reachable in testing. The audit was performed with three instruments:

1. **Upgrade Status**, which reports calls to APIs deprecated through the standard annotation mechanism, per module, with file and line references.
2. **Drupal Rector**, run in dry-run mode first, to identify which of those calls could be transformed automatically.
3. **Direct source search**, for the functions that predate the current deprecation infrastructure. Functions removed in Drupal 8 — such as `drupal_mail()`, `arg()`, `drupal_set_message()` and `language_default()` — are not in any deprecation database, because they have not existed for several major versions. Only a textual search finds them, and in these modules they were the majority of the problem.

The third instrument produced the counts already given in Table 1.3, and it is the reason the audit was worth the time it took. A migration driven only by tool output would have declared the modules nearly clean and then failed in production.

3.3.2 Order of work

The modules were sequenced deliberately rather than tackled in parallel.

FreeCAD Lab Migration was taken first. It is the module with the largest number of form classes and the highest incidence of legacy mail code, which made it the best place to work out the patterns. Every decision made there — how the mail service should be shaped, what the AJAX helper should return, how the CSV export should be routed — was made once, in a module where the consequences were visible, and then reused.

DWSIM Custom Model was taken second, because its workflow is structurally almost identical to the FreeCAD one: submit proposal, manage proposal, approve or disapprove, submit code, bulk-approve, mark complete. This similarity made it the natural first test of whether the services designed for FreeCAD were general or merely fitted to one module.

DWSIM Flowsheet and DWSIM Textbook Companion followed. Textbook Companion was deliberately left until last: it has the largest route table of the four and, at sixteen form classes, the widest surface area, so it benefited most from patterns that were by then settled.

3.3.3 Incremental delivery

All work was carried out on a dedicated branch, `sli-intern`, and pushed at the completion of each coherent unit of work rather than accumulated locally. This was a requirement set by the mentor, and its value became apparent during review: a commit that migrates one form class can be read and commented on, whereas a commit that migrates twenty cannot. Chapter 4 describes the branch and review workflow in more detail.

3.4 Success Criteria

The work was considered complete for a given module when all of the following held:

- The module installs and enables on a Drupal 11 site without errors.
- A search of the module source returns no occurrences of removed APIs in live code paths.
- Every workflow step — submission, approval, code upload, bulk approval, completion — executes and produces the same result as on the Drupal 10 site.
- Notification mail is dispatched for every event that previously dispatched mail, with the same recipients.
- No AJAX callback body is duplicated between form classes within the module.
- The changes are pushed to `sli-intern` and have been reviewed.

Chapter 4

Development Environment and Workflow

4.1 Overview

Setting up a working environment consumed a non-trivial portion of the first two weeks, and it is documented here in detail because it is the part of the work most likely to be repeated by whoever continues it. The requirement is not merely to run Drupal; it is to run two Drupal versions concurrently, since verifying that behaviour has been preserved requires having the Drupal 10 site available for comparison while the Drupal 11 site is being built.

4.2 Technologies Used

Table 4.1: Technologies used during the internship

Technology	Version	Role in the project
Drupal	10	Source platform; baseline for behavioural comparison
Drupal	11	Target platform for all migrated modules
PHP	8.3	Runtime required by Drupal 11
Composer	2.x	Dependency resolution for core and contributed modules
Drush	12/13	Cache rebuilds, database updates, module management, log inspection
MySQL	8.x	Storage for core tables and the custom module schemas
Apache	2.4	Web server serving the Drupal document root
Docker	—	Containerised development stack on the primary machine
Podman	—	Rootless container alternative used on a second machine
Git	2.x	Version control; feature branch <code>sli-intern</code>
GitHub	—	Remote hosting, pull requests and code review
JavaScript / AJAX	—	Cascading dropdowns and dynamic form regions
HTML, CSS	—	Administrative table markup and form presentation
Twig	3.x	Template layer for rendered output
YAML	—	Routing, services, permissions, menu links, configuration schema

4.3 Containerised Setup

The stack was defined as three services: a PHP-Apache container serving the Drupal document root, a MySQL container, and a database administration container for inspection. The Drupal codebase was bind-mounted from the host so that edits made in the host editor took effect immediately without rebuilding the image.

Two decisions in this setup were made after initial mistakes and are worth recording.

The container must contain the PHP extensions Drupal expects

The stock PHP image does not include `pdo_mysql`, `gd` or `zip`, all of which Drupal requires or strongly prefers. Drupal’s installer reports these as requirement failures at the very last step of installation, after the database has already been configured, which

makes the failure feel more serious than it is. The extensions were installed in the image build rather than patched into a running container, so that the environment could be recreated.

File ownership across the bind mount

Under Docker, files written by the web server inside the container are owned by the container's web user, which does not correspond to the host user. Drupal writes to `sites/default/files` during installation and whenever a contributor uploads a file, and it writes generated PHP into its own directories. The symptom of getting this wrong is an installer that cannot create the settings file, or an upload that silently fails. This was resolved by aligning the container user's UID with the host user's UID at build time.

Podman, used on the second machine, behaves differently: running rootless, it maps the container user to the host user by default through user namespaces, which avoids the problem but introduces its own — SELinux labelling on the bind mount requires the appropriate mount option, without which the container cannot read the mounted source at all. The two machines therefore needed slightly different compose definitions, and this was documented in the repository README rather than left as tribal knowledge.

4.4 Composer and the Core Upgrade

The site's dependencies were upgraded by raising the constraint on the core metapackages and letting Composer resolve:

```
1 # Inspect what currently blocks the upgrade.
2 composer why-not drupal/core-recommended 11
3
4 # Raise the constraints on core and its companions together.
5 composer require drupal/core-recommended:^11 \
6     drupal/core-composer-scaffold:^11 \
7     drupal/core-project-message:^11 \
8     --update-with-all-dependencies
9
10 # Apply pending database updates, then rebuild all caches.
11 drush updb -y
12 drush cr
```

Listing 4.1: Core upgrade sequence

The `why-not` command was the most useful single command of the entire setup phase. Composer's failure output on a large Drupal site is a wall of version constraints in which the actual blocking package is not obvious; `why-not` answers the question directly.

The blockers encountered fell into two categories. Some contributed modules had Drupal 11-compatible releases that simply had not been requested, and raising the constraint resolved them. Others had no compatible release at the time; for each of these the decision was whether the module was genuinely required by the custom workflows or was incidental. Modules in the second category were removed from the site rather

than pinned, since carrying an unmaintained dependency forward defeats the purpose of the upgrade.

4.5 Module Metadata

Before any code change could be tested, each module's `info.yml` had to admit Drupal 11. Every one of the four modules carried a constraint that excluded it:

```
1 # Before -- module is invisible to a Drupal 11 site.
2 name: 'FreeCad Lab Migration'
3 description: 'IIT Bombay Lab Migration project'
4 package: IITB
5 core_version_requirement: ^8 || ^9 || ^10
6 type: module
7
8 # After.
9 name: 'FreeCad Lab Migration'
10 description: 'IIT Bombay Lab Migration project'
11 package: IITB
12 core_version_requirement: ^10 || ^11
13 type: module
```

Listing 4.2: Module metadata before and after

Dropping `^8` and `^9` from the constraint was a conscious choice rather than an oversight. Those versions are end-of-life; advertising support for them implies a compatibility promise that nobody is in a position to verify, and it keeps alive the expectation that the code must remain runnable on APIs that have been gone for years. Retaining `^10` was necessary because the production sites were still on Drupal 10 while the migration was in progress.

4.6 Git Workflow

All work was carried out on a branch named `sli-intern`, created from the main development branch of each repository.

```
1 git checkout -b sli-intern
2 # ... work on one coherent unit: one form, one service, one feature ...
3 git add src/Services/LabMigrationMailService.php lab_migration.services.yml
4 git commit -m "Add reusable mail service and register it in the container"
5 git push origin sli-intern
```

Listing 4.3: Branch and review workflow

Three conventions were followed, all of which were mentor requirements rather than personal preference, and all of which proved their value during review:

1. **One logical change per commit.** A commit that migrates a single form class from `drupal_mail()` to the mail service can be reviewed in a minute. A commit that migrates fourteen of them cannot be reviewed at all, only approved on trust.
2. **Push frequently.** Work that exists only on a laptop is invisible to the mentor and cannot be commented on until it is finished, by which point any architectural

feedback is expensive to act on. Pushing at the end of each unit of work meant that feedback on the shape of the mail service arrived while only one module had adopted it.

3. **Never rewrite pushed history.** Once a commit is on the remote branch it is part of the review record.

The review cycle itself was the most instructive part of the process. Comments fell into three recurring categories: use of a static service call where injection was available, duplicated logic that should have been delegated to the shared service, and behaviour changes introduced inadvertently while refactoring — a notification that was dropped, or a status value written as a different integer. The third category is the one that justified the workflow-level testing described in Chapter 11.

4.7 Daily Cycle

The practical working loop, repeated many times a day, was:

1. Edit source on the host.
2. **drush cr** — rebuild caches. Required after any change to routing, services, hooks or plugin definitions, and harmless otherwise.
3. Exercise the affected screen in the browser.
4. **drush ws -count=20** — inspect the watchdog log if anything misbehaved.
5. Commit and push when the unit of work is complete.

Step 2 is the one that catches out newcomers. A newly declared route returns “page not found” until the router is rebuilt; a newly declared service throws `ServiceNotFoundException` until the container is rebuilt; a newly added `hook_mail()` implementation is not invoked until the hook registry is rebuilt. Every one of these presents as a code error and none of them is one. A substantial fraction of the debugging time lost in the first fortnight was attributable to this single cause, and the habit of rebuilding caches reflexively before diagnosing anything eliminated it.

Chapter 5

Drupal 11 Migration and Deprecated API Review

5.1 Migration Pipeline

Each module was taken through the same five-stage pipeline shown in Figure 5.1. The stages are ordered by dependency: nothing can be tested until the module installs, the module cannot install until its metadata admits Drupal 11, and metadata alone is useless if the code fatally errors on the first request.

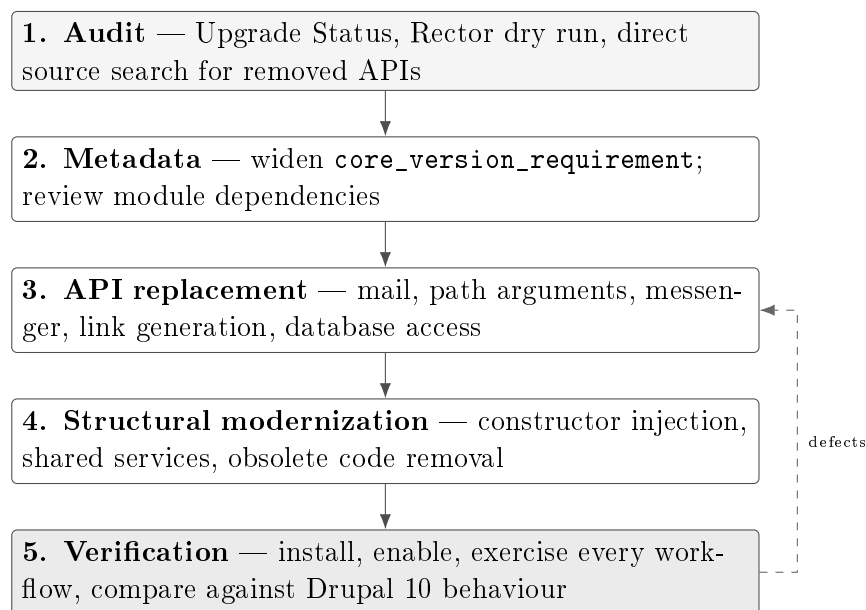


Figure 5.1: Five-stage migration pipeline applied to each module

5.2 Inventory of Deprecated and Removed APIs

Table 5.1 is the consolidated inventory produced by the audit. It is the working document that drove the migration, and each row is discussed in the sections that follow.

Table 5.1: Deprecated and removed APIs found, and their replacements

Legacy API	Removed in	Replacement	Nature of change
<code>drupal_mail()</code>	Drupal 8	<code>plugin.manager.mail</code> service, <code>->mail()</code>	Semantic: language argument type changed
<code>language_default()</code>	Drupal 8	<code>language_manager</code> service, <code>->getDefaultLanguage()</code> <code>->getId()</code>	Semantic: object replaced by langcode string
<code>arg(\$n)</code>	Drupal 8	Route parameter injected into the method signature	Semantic: requires knowing the route
<code>drupal_set_message()</code>	Drupal 9	<code>messenger</code> service, <code>->addStatus()</code> , <code>->addError()</code>	Mechanical, but severity maps to distinct methods
<code>\Drupal::l()</code>	Drupal 9	<code>Link::fromTextAndUrl()</code> with a <code>Url</code> object	Mechanical
<code>db_query()</code> , <code>db_select()</code>	Drupal 9	<code>database</code> service query builder	Mechanical; present mainly in comments
<code>drupal_get_path()</code>	Drupal 10	<code>extension.list.module</code> service, <code>->getPath()</code>	Mechanical
<code>REQUEST_TIME</code>	Drupal 10	<code>datetime.time</code> service, <code>->getRequestTime()</code>	Mechanical
<code>global \$user</code>	Drupal 8	<code>current_user</code> service	Semantic: proxy object, not a full user entity
Procedural AJAX call-backs	Drupal 8	Methods on the form class, referenced as <code>::method</code>	Structural

5.3 Mail API Migration

The single most consequential replacement was `drupal_mail()`, which appeared 65 times across the four modules. Chapter 7 describes the reusable mail service that these call sites were eventually routed through; this section covers the API-level transformation itself.

The legacy call, as found in `LabMigrationSolutionProposalApprovalForm.php`, looked like this:

```
1 if (!drupal_mail('lab_migration', 'solution_proposal_approved', $email_to,
2     language_default(), $param, $from, TRUE)) {
3     drupal_set_message(t('Error sending email message.'), 'error');
4 }
```

Listing 5.1: Legacy mail dispatch, as inherited

Three separate removed APIs appear in these three lines: `drupal_mail()`, `language_default()` and `drupal_set_message()`. The rewritten form is:

```
1 $langcode = $this->languageManager->getDefaultLanguage()->getId();
2 $result = $this->mailManager->mail(
3     'lab_migration',
4     'solution_proposal_approved',
5     $email_to,
6     $langcode,
7     $param,
8     $from,
9     TRUE
10 );
11
12 if (empty($result['result'])) {
13     $this->messenger->addError($this->t('Error sending email message.'));
14     $this->logger->error('Mail @key to @to failed.',
```

```

15     ['@key' => 'solution_proposal_approved', '@to' => $email_to]);
16 }

```

Listing 5.2: Migrated mail dispatch

Two details deserve comment.

First, the return value. `drupal_mail()` returned a boolean; `MailManagerInterface::mail()` returns the message array, whose `'result'` key holds the boolean. Code written as `if (!$mailManager->mail(...))` therefore never enters the error branch, because a non-empty array is always truthy. This is a silent failure: mail stops being delivered and nothing reports it. Several call sites had been converted in exactly this way by earlier partial migrations, and correcting them was one of the more valuable outcomes of the audit.

Second, the logger. The original code reported mail failure only as a message on screen, which is invisible when mail is dispatched from a batch or bulk-approval operation. Adding a logger channel means failures are recoverable from `drush ws` after the fact.

5.4 Path Argument Migration

The `arg()` function returned a segment of the current path by index. In Drupal 7, where routing was performed by `hook_menu()` and path inspection, this was the normal way for a form to discover which record it was operating on. The function does not exist in Drupal 8 or later, and 155 occurrences of it were found across the four modules — some live, most commented out beside a replacement that had been applied earlier.

The modern equivalent is not a function call at all. Route parameters declared in the routing file are passed as method arguments:

```

1 lab_migration.proposal_status_form:
2   path: /lab-migration/manage-proposal/status/{id}
3   defaults:
4     _title: 'Proposal Status'
5     _form: \Drupal\lab_migration\Form\LabMigrationProposalStatusForm
6   requirements:
7     _permission: 'lab migration approve proposal'

```

Listing 5.3: Route with a parameter

```

1 // Before: positional, brittle, and dependent on the URL structure.
2 $proposal_id = (int) arg(3);
3
4 // After: the router supplies the parameter by name.
5 public function buildForm(array $form, FormStateInterface $form_state,
6                           $id = NULL) {
7     $proposal_id = (int) $id;
8     // ...
9 }

```

Listing 5.4: Path argument access before and after

The improvement is not merely that the code compiles. With `arg(3)` the form is coupled to the exact depth of the path: moving the route from `/lab-migration/`

`manage-proposal/status/{id}` to `/lab-migration/status/{id}` silently changes which segment index 3 refers to, and the form begins reading the wrong value without any error. With a named route parameter the URL can be restructured freely.

The migration of these call sites was the least mechanical part of the whole exercise, because it required, for each form, identifying which route or routes reach it and confirming that the parameter name matches. Forms reachable from more than one route — the certificate forms, which are reached with different path prefixes for different certificate types — required their routes to be harmonised so that a single parameter name applied to all of them.

5.5 Messenger and Link Generation

Two smaller categories were mechanical but numerous.

`drupal_set_message()` took the message and a severity string. The messenger service exposes a distinct method per severity, which is a genuine improvement in readability:

```
1 // Before.
2 drupal_set_message(t('Invalid proposal selected. Please try again.'), 'error');
3 drupal_set_message(t('Proposal approved.'));
4
5 // After.
6 $this->messenger()->addError($this->t('Invalid proposal selected. Please try again.'));
7 $this->messenger()->addStatus($this->t('Proposal approved.'));
```

Listing 5.5: Messenger migration

Link generation via `\Drupal::l()` was replaced with the `Link` and `Url` value objects. The controller contained calls of the form:

```
1 // Before.
2 '#markup' => \Drupal::l($proposal_data->lab_title,
3                       'lab-migration/proposal/' . $proposal_data->id),
4
5 // After.
6 '#markup' => Link::fromTextAndUrl(
7   $proposal_data->lab_title,
8   Url::fromRoute('lab_migration.proposal_view', ['id' => $proposal_data->id])
9 )->toString(),
```

Listing 5.6: Link generation migration

Routing by name rather than by literal path is the substantive gain here. A route name is stable across path changes, and a mistyped route name fails loudly at render time rather than producing a link that quietly 404s.

5.6 Database Access

The modules had already moved off `db_query()` in live code; what remained were the original procedural queries preserved as comments next to their replacements. These were removed rather than retained. The reasoning: a commented Drupal 7 query has no

informational value that the working query does not also carry, and it invites a future maintainer to restore an approach that cannot work.

Live query code was left substantially as it was, since the query builder API has been stable, but two adjustments were made. Static `Database::getConnection()` calls inside form classes were replaced with the injected `database` service, for the reasons given in Section 5.7. And placeholder usage was reviewed for correctness — the query builder parameterises conditions automatically, but a handful of places constructed literal SQL fragments through string concatenation, and those were rewritten to use the builder’s condition methods.

5.7 Dependency Injection

The audit counted well over a thousand static service calls across the four modules. Converting all of them was neither necessary for Drupal 11 compatibility nor achievable in the available time, so a rule was applied: any class touched for another reason had its dependencies converted; classes not otherwise touched were left alone.

For form and controller classes, injection is performed through the `create()` factory:

```
1 class LabMigrationProposalApprovalForm extends FormBase {
2
3   protected $mailService;
4   protected $database;
5   protected $currentUser;
6
7   public function __construct(LabMigrationMailService $mail_service,
8                               Connection $database,
9                               AccountProxyInterface $current_user) {
10     $this->mailService = $mail_service;
11     $this->database = $database;
12     $this->currentUser = $current_user;
13   }
14
15   public static function create(ContainerInterface $container) {
16     return new static(
17       $container->get('lab_migration.mail_service'),
18       $container->get('database'),
19       $container->get('current_user')
20     );
21   }
22 }
```

Listing 5.7: Constructor injection in a form class

For plain service classes, the dependencies are declared in the service definition and the container passes them to the constructor:

```
1 services:
2   lab_migration.mail_service:
3     class: Drupal\lab_migration\Services\LabMigrationMailService
4     arguments:
5       - '@plugin.manager.mail'
6       - '@language_manager'
7       - '@config.factory'
8       - '@logger.factory'
9
10  lab_migration.ajax_helper:
11    class: Drupal\lab_migration\Services\AjaxHelper
```

```
12     arguments:
13         - '@database'
```

Listing 5.8: Service definitions with arguments

The contrast with the inherited service file is instructive. As found, it declared service classes with no arguments at all, and those classes then reached into the container themselves — one of them attempting to call `$this::database()` as though `FormBase`'s helper were available on an arbitrary class, which is the kind of error that only surfaces when the branch containing it is executed.

5.8 Removal of Obsolete Code

A deliberate cleanup pass removed:

- Commented-out Drupal 6 and Drupal 7 query and mail code, superseded by working replacements.
- `var_dump($x);die;` debugging statements left in the source, several of which sat in branches that were rarely executed and would have halted the request had they been reached.
- Unused `use` statements, including imports of classes that do not exist — `Drupal\Core\LabMigration\PluginMail` was imported by the mail service class and has never existed in any Drupal version.
- Functions that were unreachable because the routes referencing them had already been removed.

Each removal was made in its own commit, separate from behavioural changes, so that a reviewer could confirm at a glance that nothing functional was being altered.

5.9 Outcome

Table 5.2 summarizes the state of each module after migration.

Table 5.2: Migration summary by module

Module	Core constraint	con-	Installs D11	on	Principal changes
FreeCAD Lab Migra- tion	~10		~11	Yes	Mail, AJAX, DI, CSV ex- port, path arguments Mail service, AJAX helper, CSV export, work- flow verification Mail service, approval workflow, deprecated code removal Forms, controllers, ser- vices, routing, mail ser- vice, AJAX consolidation
DWSIM Custom Model	~10		~11	Yes	
DWSIM Flowsheet	~10		~11	Yes	
DWSIM Textbook Companion	~10		~11	Yes	

Chapter 6

Reusable AJAX Architecture

6.1 The Pattern Being Implemented

Almost every non-trivial screen in these applications uses the same interaction pattern: a chain of dependent dropdowns. In the FreeCAD Lab Migration module the chain is lab → experiment → solution → solution files. In the DWSIM modules it is proposal → model → submitted code, or book → chapter → example. The mechanics are identical: selecting a value in one dropdown triggers an AJAX request; the server queries the database for the children of the selected record; the dependent dropdown is rebuilt with those options; and the region of the page containing it is replaced.

Figure 6.1 shows the request cycle.

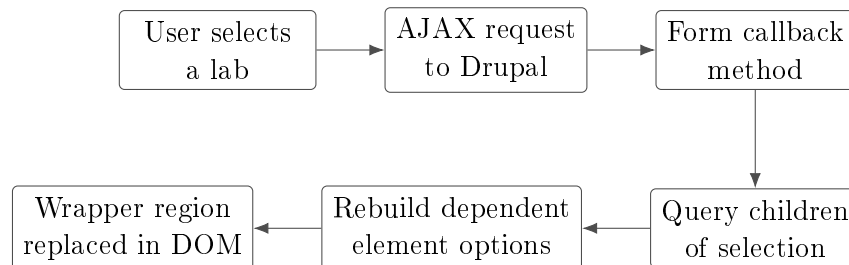


Figure 6.1: AJAX request cycle for a cascading dropdown

6.2 The Architecture as Inherited

Because the pattern recurs, so did its implementation. The AJAX logic in each form consisted of two kinds of method, and both were duplicated.

6.2.1 Duplicated callback methods

The callback method itself is trivial — it returns the part of the form that should replace the wrapper. In `LabMigrationRunForm.php` it appears as:

```

1 public function ajax_experiment_list_callback(array &$form,
2                                             FormStateInterface $form_state) {
3     return $form['download_lab_wrapper'];
4 }
5
6 public function ajax_solution_list_callback(array &$form,
7                                             FormStateInterface $form_state) {
8     return $form['download_experiment_wrapper'];
9 }
10
11 public function ajax_solution_files_callback(array &$form,
12                                             FormStateInterface $form_state) {
13     return $form['download_solution_wrapper'];
14 }

```

Listing 6.1: AJAX callbacks in LabMigrationRunForm

The same three methods, with the same bodies and near-identical names, appear in LabMigrationBulkApprovalForm.php:

```

1 public function ajax_experiment_list_callback(array &$form,
2                                             FormStateInterface $form_state) {
3     return $form['download_lab_wrapper'];
4 }
5
6 public function ajax_solution_list_callback(array &$form,
7                                             FormStateInterface $form_state) {
8     return $form['download_experiment_wrapper'];
9 }
10
11 public function ajax_solution_file_callback(array &$form,
12                                             FormStateInterface $form_state) {
13     return $form['download_solution_wrapper'];
14 }

```

Listing 6.2: The same callbacks in LabMigrationBulkApprovalForm

Note the third method name: `ajax_solution_file_callback` in one form and `ajax_solution_files_callback` in the other. The bodies are identical; only the name has drifted. This is exactly the failure mode that duplication produces — two copies that were once the same, edited independently, now differing in a way that carries no meaning but must nevertheless be respected by anyone reading either file.

6.2.2 Duplicated option builders

More substantial duplication lay in the methods that build the dropdown options. These execute the actual database queries, and they too were copied between forms — `_ajax_get_experiment_list()` in one form and `_ajax_bulk_get_experiment_list()` in another, differing only in incidental details such as whether an approval-status filter was applied.

```

1 public function _ajax_bulk_get_experiment_list($lab_default_value = '') {
2     $experiments = ['0' => 'Please select...'];
3     $connection = Database::getConnection();
4     $query = $connection->select('lab_migration_experiment', 'e');
5     $query->fields('e', ['id', 'number', 'title']);
6     $query->condition('proposal_id', $lab_default_value);
7     $query->orderBy('number', 'ASC');
8     foreach ($query->execute() as $row) {
9         $experiments[$row->id] = $row->number . ' . ' . $row->title;

```

```

10 }
11 return $experiments;
12 }

```

Listing 6.3: Option builder duplicated across forms (abridged)

Table 6.1 quantifies the situation across the four modules at the start of the internship.

Table 6.1: AJAX implementation surface before consolidation

Module	'#ajax' attachments	Callback methods	Forms
FreeCAD Lab Migration	6	11	3
DWSIM Textbook Companion	7	4	3
DWSIM Custom Model	4	2	2
DWSIM Flowsheet	15	3	3
Total	32	20	11

6.2.3 Why this mattered

Three concrete costs were attributable to the duplication.

Divergence. Two copies of an option builder that were once identical had drifted: one filtered experiments by approval status and the other did not. Which behaviour is correct depends on the form, so this may have been intentional — but nothing in the code says so, and a maintainer reading either copy cannot tell whether the difference is a design decision or a bug.

Cost of change. A request to change how experiment titles are formatted in drop-downs — prefixing the experiment number, say — required editing every copy. Missing one produces an inconsistency that is visible to users and invisible to the developer who made the change.

Untestability. A query embedded in a form method can only be exercised by rendering the form and triggering an AJAX request. The same query in a service can be called directly.

6.3 The AjaxHelper Service

The consolidation separates the two responsibilities that were tangled together.

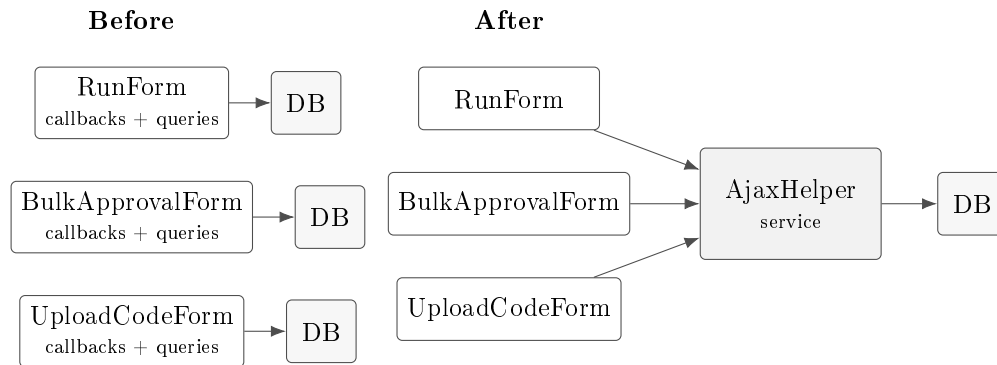


Figure 6.2: AJAX option-building before and after consolidation

6.3.1 Design decisions

Three decisions shaped the service, and each involved rejecting a plausible alternative.

The service builds options; it does not return render arrays. A tempting design has the service return the entire AJAX response, including the form fragment. This was rejected because the fragment is form-specific — it depends on which wrapper key the form uses — while the query is not. Keeping the service free of render-array structure is what makes it reusable across forms whose element trees differ.

The callback methods remain on the form. They are one line each, and moving them to a shared base class would achieve little while introducing an inheritance dependency. What the consolidation removes is not the callback but the query logic behind it. Where forms did need common behaviour, a trait was used in preference to a base class, since Drupal form classes already extend `FormBase` and the inheritance slot is taken.

Options are returned keyed by record ID, with a leading prompt. This preserves the existing convention exactly, so that submit handlers reading `$form_state->getValue()` continue to receive the same values they received before. Preserving the value contract was what allowed the forms to be converted one at a time rather than in a single flag-day change.

6.3.2 Implementation

```

1 namespace Drupal\lab_migration\Services;
2
3 use Drupal\Core\Database\Connection;
4
5 /**
6  * Builds option lists for the cascading dropdowns used across the module.
7  */
8 class AjaxHelper {
9
10     protected $database;
11
12     public function __construct(Connection $database) {
13         $this->database = $database;
14     }
15

```

```

16  /**
17   * Experiments belonging to a proposal.
18   *
19   * @param int $proposal_id
20   * The proposal whose experiments are wanted.
21   * @param bool $approved_only
22   * Restrict to approved experiments. Made explicit because the two
23   * copies of this logic that previously existed disagreed on it.
24   */
25  public function getExperimentOptions($proposal_id, $approved_only = FALSE) {
26      $options = ['0' => 'Please select...'];
27      if (empty($proposal_id)) {
28          return $options;
29      }
30
31      $query = $this->database->select('lab_migration_experiment', 'e')
32          ->fields('e', ['id', 'number', 'title'])
33          ->condition('e.proposal_id', $proposal_id)
34          ->orderBy('e.number', 'ASC');
35
36      if ($approved_only) {
37          $query->condition('e.approval_status', 1);
38      }
39
40      foreach ($query->execute() as $row) {
41          $options[$row->id] = $row->number . ' . ' . $row->title;
42      }
43      return $options;
44  }
45
46  public function getSolutionOptions($experiment_id) { /* ... */ }
47
48  public function getSolutionFileOptions($solution_id) { /* ... */ }
49
50  public function getLabOptions($only_displayable = TRUE) { /* ... */ }
51  }

```

Listing 6.4: The AjaxHelper service

The `$approved_only` parameter is where the divergence discovered during the audit was resolved. Rather than picking one behaviour and imposing it on both call sites — which would silently change one of the screens — the difference was made an explicit argument, and each call site passes the value corresponding to what it did before. The difference is now visible in the call rather than buried in a duplicated method body.

6.3.3 Use from a form

```

1  public static function create(ContainerInterface $container) {
2      return new static($container->get('lab_migration.ajax_helper'));
3  }
4
5  public function buildForm(array $form, FormStateInterface $form_state) {
6      $form['lab'] = [
7          '#type' => 'select',
8          '#title' => $this->t('Title of the lab'),
9          '#options' => $this->ajaxHelper->getLabOptions(),
10         '#ajax' => [
11             'callback' => '::ajaxExperimentListCallback',
12             'wrapper' => 'ajax_selected_lab',
13         ],
14     ];
15
16     $form['download_lab_wrapper'] = [
17         '#type' => 'container',

```

```

18     '#attributes' => ['id' => 'ajax_selected_lab'],
19 ];
20
21 $lab = $form_state->getValue('lab');
22 if (!empty($lab)) {
23     $form['download_lab_wrapper']['lab_experiment_list'] = [
24         '#type' => 'select',
25         '#title' => $this->t('Title of the experiment'),
26         '#options' => $this->ajaxHelper->getExperimentOptions($lab),
27         '#ajax' => [
28             'callback' => '::ajaxSolutionListCallback',
29             'wrapper' => 'ajax_download_experiments',
30         ],
31     ];
32 }
33 return $form;
34 }
35
36 public function ajaxExperimentListCallback(array &$form,
37                                           FormStateInterface $form_state) {
38     return $form['download_lab_wrapper'];
39 }

```

Listing 6.5: A form using the helper

Callback methods were also renamed from the `snake_case` convention inherited from Drupal 7 to the `camelCase` convention that Drupal’s coding standards require for class methods. This is cosmetic in isolation, but it is the difference between code that reads as a modern Drupal class and code that reads as a mechanically converted Drupal 7 form.

6.4 Results

Table 6.2: Effect of AJAX consolidation

Property	Before	After
Places where experiment options are queried	4	1
Places where solution options are queried	3	1
Callback methods with duplicated bodies	11	0
Naming of equivalent callbacks	Inconsistent	Consistent
Approval-status filtering	Implicit, divergent	Explicit parameter
Query logic reachable without rendering a form	No	Yes

The benefit that mattered most in practice was not the reduction in line count. It was that a subsequent change to how solution files are listed — required for the bulk approval screen — was made in exactly one method and took effect on every screen that lists solution files, with no possibility of having missed a copy.

Chapter 7

Centralized Mail Service

7.1 Why Mail Matters Here

Notification mail is not a peripheral feature of these applications; it is how the workflow advances. A contributor who submits a proposal learns that it was received by email. A reviewer learns that a proposal is awaiting attention by email. An approval, a rejection, a request for revision, a code submission and a completion are all communicated by email. If mail fails silently, the workflow does not stop — it simply stalls, with each party waiting for the other, and nothing in the application indicates that anything is wrong.

This is what makes the state of the inherited mail code serious rather than merely untidy.

7.2 The Situation as Inherited

7.2.1 A removed API, called 65 times

Every module dispatched mail through `drupal_mail()`, a Drupal 7 function removed in Drupal 8. Table 7.1 gives the distribution.

Table 7.1: Legacy mail call sites found during the audit

Module	<code>drupal_mail()</code>	<code>language_default()</code>	Files affected
FreeCAD Lab Migration	11	15	5
DWSIM Textbook Companion	44	44	9
DWSIM Custom Model	7	4	4
DWSIM Flowsheet	3	3	3
Total	65	66	21

7.2.2 Message composition scattered across forms

Beyond the API problem, the mail logic was structurally misplaced. Recipient resolution, sender configuration lookup, parameter assembly and error handling were performed inside form submit handlers, repeated in each handler that sent mail. A representative example, drawn from the solution proposal approval form, shows all four concerns interleaved with workflow logic:

```
1 $config = \Drupal::config('lab_migration.settings');
2 $from = $config->get('lab_migration_from_email', NULL);
3 $email_to = $proposal_data->email;
4 $param['solution_proposal_approved']['proposal_id'] = $proposal_id;
5 $param['solution_proposal_approved']['user_id'] = $proposal_data->uid;
6 $param['solution_proposal_approved']['headers'] = ['Cc' => $cc_email];
7
8 if (!drupal_mail('lab_migration', 'solution_proposal_approved', $email_to,
9                 language_default(), $param, $from, TRUE)) {
10   drupal_set_message(t('Error sending email message.'), 'error');
11 }
```

Listing 7.1: Mail logic embedded in a submit handler

Three observations follow from code shaped like this.

First, changing the sender address — a routine administrative request — requires editing every one of the twenty-one files that read it independently.

Second, the header conventions drift. Some call sites set a `Cc` header, others do not, and whether that reflects a decision or an omission is not recoverable from the code.

Third, and most seriously, the error branch is unreachable in any call site that was partially modernized. As noted in Chapter 5, `MailManagerInterface::mail()` returns an array rather than a boolean, so `if (!$mailManager->mail(...))` is never true. Mail that fails to send reports nothing at all.

7.2.3 The cross-module requirement

A specific requirement made the case for a service rather than a base class: the FreeCAD Lab Migration module needed to invoke a mail routine that lives in a different module. Inheritance cannot express this — a form in one module cannot extend a class in another without creating a hard module dependency — but the service container can, because a service is addressable by name from anywhere in the application once its provider module is enabled.

7.3 The MailService Design

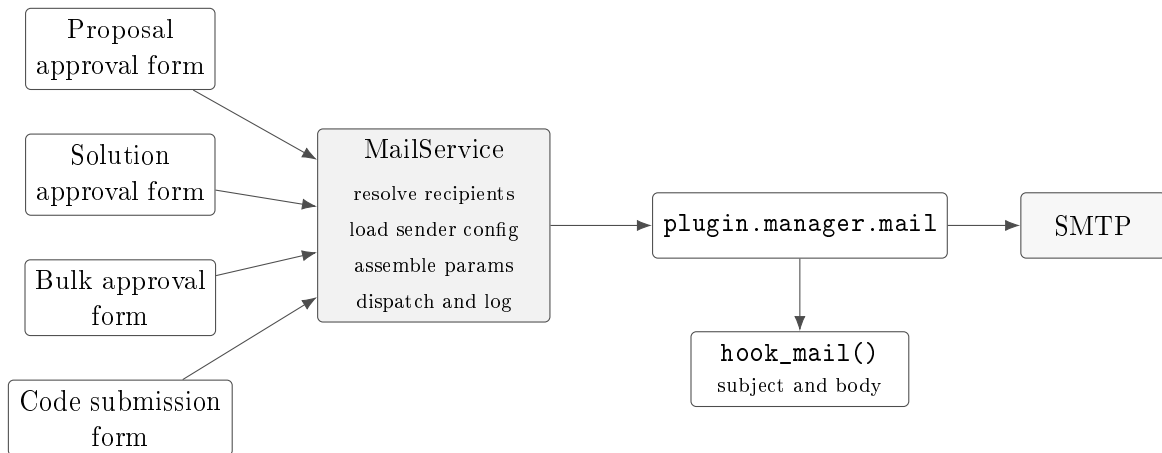


Figure 7.1: MailService architecture: all dispatch flows through one component

7.3.1 Responsibilities

The service owns exactly those concerns that were previously repeated:

- Resolving the sender address from module configuration, once.
- Resolving the default language code in the form the mail manager expects.
- Assembling the parameter array in the structure `hook_mail()` expects for each key.
- Dispatching through the mail plugin manager.
- Interpreting the result correctly and logging failures to a named channel.

It deliberately does not own message content. Subject and body remain in `hook_mail()`, keyed by message key, because that is the extension point Drupal defines for the purpose and because it is where translation and text overrides are expected to live. Moving content into the service would have produced a component that was locally convenient and inconsistent with every other Drupal module on the site.

7.3.2 Implementation

```
1 namespace Drupal\lab_migration\Services;
2
3 use Drupal\Core\Config\ConfigFactoryInterface;
4 use Drupal\Core\Language\LanguageManagerInterface;
5 use Drupal\Core\Logger\LoggerChannelFactoryInterface;
6 use Drupal\Core\Mail\MailManagerInterface;
7
8 class LabMigrationMailService {
9
10     protected $mailManager;
11     protected $languageManager;
12     protected $configFactory;
13     protected $logger;
14 }
```

```

15 public function __construct(MailManagerInterface $mail_manager,
16                             LanguageManagerInterface $language_manager,
17                             ConfigFactoryInterface $config_factory,
18                             LoggerChannelFactoryInterface $logger_factory) {
19     $this->mailManager = $mail_manager;
20     $this->languageManager = $language_manager;
21     $this->configFactory = $config_factory;
22     $this->logger = $logger_factory->get('lab_migration');
23 }
24
25 /**
26  * Dispatch a notification.
27  *
28  * @param string $key
29  * Message key handled by hook_mail().
30  * @param string $to
31  * Recipient address.
32  * @param array $params
33  * Data for hook_mail(), keyed by message key.
34  * @param array $headers
35  * Optional additional headers, e.g. Cc.
36  *
37  * @return bool
38  * TRUE if the mail system accepted the message.
39  */
40 public function send($key, $to, array $params = [], array $headers = []) {
41     if (empty($to)) {
42         $this->logger->warning('Mail @key skipped: no recipient.',
43                               ['@key' => $key]);
44         return FALSE;
45     }
46
47     $config = $this->configFactory->get('lab_migration.settings');
48     $from = $config->get('lab_migration_from_email');
49     $langcode = $this->languageManager->getDefaultLanguage()->getId();
50
51     if (!empty($headers)) {
52         $params[$key]['headers'] = $headers;
53     }
54
55     $message = $this->mailManager->mail(
56         'lab_migration', $key, $to, $langcode, $params, $from, TRUE
57     );
58
59     // The mail manager returns the message array; the boolean outcome is
60     // in its 'result' key. Testing the array itself is always truthy and
61     // is the reason earlier partial migrations lost their error handling.
62     if (empty($message['result'])) {
63         $this->logger->error('Mail @key to @to failed.',
64                             ['@key' => $key, '@to' => $to]);
65         return FALSE;
66     }
67
68     $this->logger->info('Mail @key sent to @to.',
69                      ['@key' => $key, '@to' => $to]);
70     return TRUE;
71 }
72 }

```

Listing 7.2: The reusable mail service

7.3.3 Registration

```

1 services:
2   lab_migration.mail_service:
3     class: Drupal\lab_migration\Services\LabMigrationMailService

```

```

4     arguments:
5         - '@plugin.manager.mail'
6         - '@language_manager'
7         - '@config.factory'
8         - '@logger.factory'

```

Listing 7.3: Service registration

7.3.4 The resulting call site

```

1  $this->mailService->send(
2      'solution_proposal_approved',
3      $proposal_data->email,
4      ['solution_proposal_approved' => [
5          'proposal_id' => $proposal_id,
6          'user_id' => $proposal_data->uid,
7      ]],
8      ['Cc' => $cc_email]
9  );

```

Listing 7.4: Mail dispatch after refactoring

Eleven lines become five, and more importantly the five that remain say what is being communicated to whom. Sender configuration, language resolution, result interpretation and logging have all moved to a place where they exist once.

7.4 Cross-Module Integration

The requirement that the Lab Migration module invoke the mail routine belonging to another module is satisfied without any structural coupling. The consuming module requests the providing module's service by name:

```

1  public static function create(ContainerInterface $container) {
2      return new static(
3          // Service provided by a different module; resolved by name from the
4          // container, with no class-level dependency between the two modules.
5          $container->get('dwsim_notification.mail_service'),
6          $container->get('database')
7      );
8  }
9
10 public function submitForm(array &$form, FormStateInterface $form_state) {
11     // ... workflow logic ...
12     $this->mailService->send('proposal_received', $email, $params);
13 }

```

Listing 7.5: Consuming a mail service defined in another module

Two properties of this arrangement are worth stating explicitly, because they are what make the service container the right mechanism rather than merely a convenient one.

The consuming class depends on an interface-shaped collaborator supplied at construction time; it does not name the providing module anywhere except in `create()`. Substituting a different mail implementation — for testing, or because the notification module is replaced — requires changing one line.

And the failure mode is loud. If the providing module is not enabled, the container throws `ServiceNotFoundException` when the form is constructed, which appears immediately and unambiguously in the log. The alternative arrangement — calling a function defined in another module’s `.module` file — fails with an undefined-function fatal error at the moment the mail is sent, which is to say in the middle of an approval operation, after the database has already been updated.

7.5 Results

Table 7.2: Effect of mail centralization

Property	Before	After
Calls to removed <code>drupal_mail()</code>	65	0
Files containing mail dispatch logic	21	4 (one per module)
Places reading the sender configuration	21	1 per module
Failure detection	Broken or absent	Logged per attempt
Header conventions	Divergent	Uniform
Cross-module invocation	Not possible	Supported

An unplanned benefit emerged from the logging. Because every dispatch now records an entry, `drush ws` became a diagnostic tool for the workflow itself: a report that a contributor never received an approval notice can be checked directly against the log, which distinguishes “the application never sent it” from “the mail server did not deliver it”. Before this change, that question was unanswerable.

Chapter 8

Export CSV Implementation

8.1 Requirement

The administrative screens present proposal data as HTML tables — pending proposals, pending solution proposals, all proposals, and so on. The data is visible but not extractable. Any activity that requires working with the data outside the browser — counting approvals in a given year for a progress report, compiling the list of participating institutions, cross-checking contributors against certificate records — had to be done by reading the screen and retyping.

The requirement, as stated by the mentor, was straightforward: give administrators a way to export proposal data, with filters for approval status and year of submission. Everything else about the implementation was left open, which made this the part of the internship with the most design latitude — and, being a new feature built on freshly migrated foundations, a useful test of whether those foundations were sound.

8.2 Design Decisions

8.2.1 A route on the existing administrative screen

The export was added as a tab alongside the existing proposal-management tabs rather than as a separate administrative page. Administrators already work on the proposal management screen; placing the export where the data is means it is discoverable without documentation. Figure 8.1 shows the tab in position on the All Proposals screen.

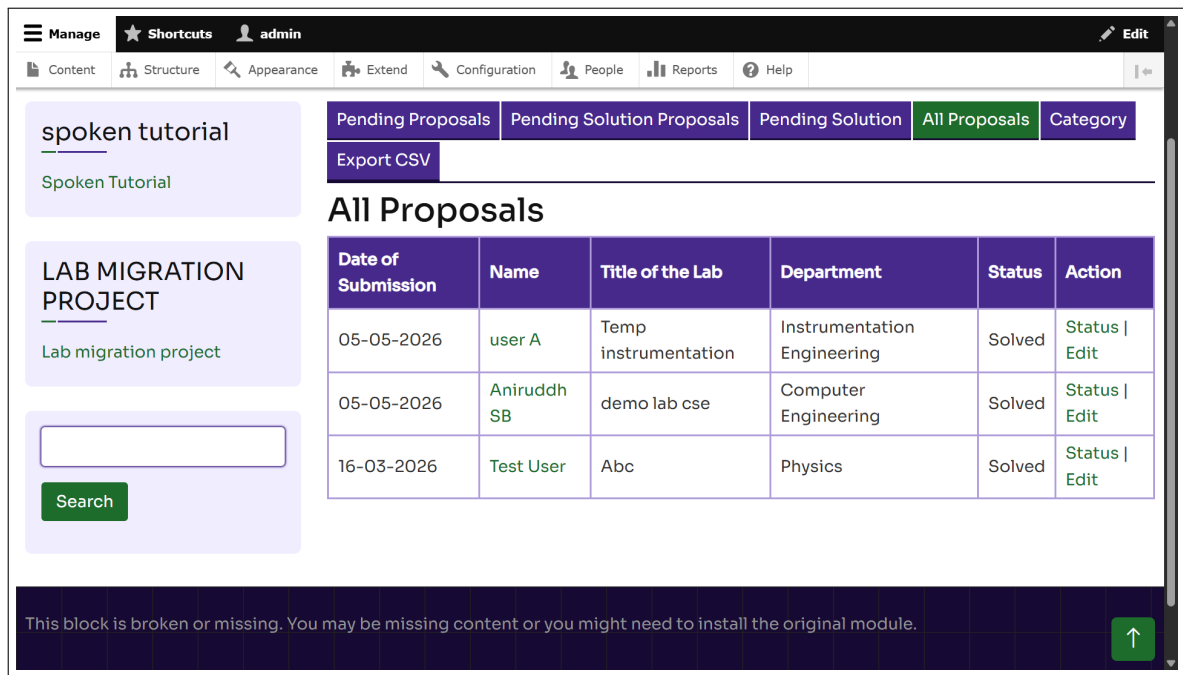


Figure 8.1: The All Proposals administrative screen, with the Export CSV tab added alongside the existing management tabs

8.2.2 A form, not a link

The export could have been implemented as a plain link that dumps everything. It was implemented as a form with filter controls instead, for two reasons. The requirement explicitly asked for filtering by status and year, and an unfiltered export of a table that accumulates every proposal ever submitted grows less useful each year — the recipient’s first action would be to filter it in a spreadsheet.

8.2.3 Filter values derived from the data

The year dropdown is populated from the distinct years actually present in the proposal table, not from a hard-coded range. A hard-coded range is wrong twice: it offers years in which nothing was submitted, and it must be edited every January. Deriving the options from a `DISTINCT` query means the control is correct without maintenance.

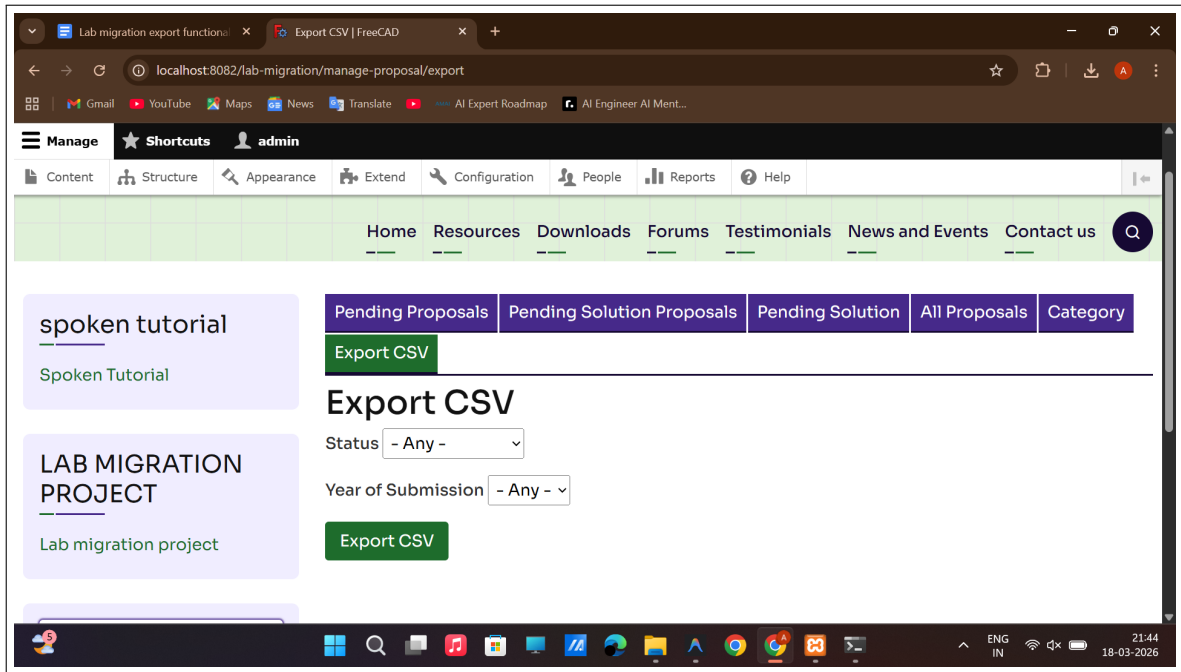


Figure 8.2: The Export CSV interface with Status and Year of Submission filters, both defaulting to *Any*

8.2.4 Both filters default to Any

Neither filter is mandatory. An administrator who wants everything selects nothing and submits. This keeps the common case — a full export — to a single click, while leaving the filters available for the specific case.

8.3 Implementation

8.3.1 Routing

```

1 # lab_migration.routing.yml
2 lab_migration.proposal_export:
3   path: /lab-migration/manage-proposal/export
4   defaults:
5     _title: 'Export CSV'
6     _form: \Drupal\lab_migration\Form\LabMigrationExportCsvForm
7   requirements:
8     _permission: 'lab migration manage proposal'
9
10 # lab_migration.links.task.yml -- places the tab beside the existing ones.
11 lab_migration.proposal_export_tab:
12   route_name: lab_migration.proposal_export
13   title: 'Export CSV'
14   base_route: lab_migration.proposal_all
15   weight: 10

```

Listing 8.1: Route and local task definition for the export

Reusing the existing `lab migration manage proposal` permission was deliberate.

The export exposes no information that the management screens do not already show to the same audience, so introducing a separate permission would have added an administrative decision without adding a security boundary.

8.3.2 The form

```

1 class LabMigrationExportCsvForm extends FormBase {
2
3   protected $database;
4
5   public static function create(ContainerInterface $container) {
6     return new static($container->get('database'));
7   }
8
9   public function getFormId() {
10    return 'lab_migration_export_csv_form';
11  }
12
13  public function buildForm(array $form, FormStateInterface $form_state) {
14    $form['status'] = [
15      '#type' => 'select',
16      '#title' => $this->t('Status'),
17      '#options' => [
18        '' => $this->t('- Any -'),
19        '0' => $this->t('Pending'),
20        '1' => $this->t('Approved'),
21        '2' => $this->t('Disapproved'),
22        '3' => $this->t('Completed'),
23      ],
24      '#default_value' => '',
25    ];
26
27    $form['year'] = [
28      '#type' => 'select',
29      '#title' => $this->t('Year of Submission'),
30      '#options' => ['' => $this->t('- Any -')] + $this->getYearOptions(),
31      '#default_value' => '',
32    ];
33
34    $form['actions']['submit'] = [
35      '#type' => 'submit',
36      '#value' => $this->t('Export CSV'),
37    ];
38
39    return $form;
40  }
41
42  /**
43   * Years for which at least one proposal exists.
44   */
45  protected function getYearOptions() {
46    $years = [];
47    $query = $this->database->select('lab_migration_proposal', 'p');
48    $query->addExpression('DISTINCT YEAR(FROM_UNIXTIME(p.creation_date))', 'yr');
49    $query->orderBy('yr', 'DESC');
50    foreach ($query->execute() as $row) {
51      if (!empty($row->yr)) {
52        $years[$row->yr] = $row->yr;
53      }
54    }
55    return $years;
56  }
57 }

```

Listing 8.2: Export form with data-derived filter options

8.3.3 Query construction

Filters are applied conditionally, so that an unselected filter contributes no condition rather than a condition that matches everything:

```
1 protected function fetchRows($status, $year) {
2     $query = $this->database->select('lab_migration_proposal', 'p')
3         ->fields('p', ['name', 'email', 'university', 'lab_title',
4             'approval_status', 'creation_date'])
5         ->orderBy('p.creation_date', 'DESC');
6
7     if ($status !== '' && $status !== NULL) {
8         $query->condition('p.approval_status', $status);
9     }
10
11     if (!empty($year)) {
12         // Half-open interval on the indexed timestamp column, rather than
13         // YEAR(FROM_UNIXTIME(...)) in the WHERE clause, which would prevent
14         // the index from being used.
15         $start = mktime(0, 0, 0, 1, 1, (int) $year);
16         $end = mktime(0, 0, 0, 1, 1, (int) $year + 1);
17         $query->condition('p.creation_date', $start, '>=');
18         $query->condition('p.creation_date', $end, '<');
19     }
20
21     return $query->execute()->fetchAll();
22 }
```

Listing 8.3: Conditional filter application

The comment records a decision worth preserving. Filtering with `YEAR(FROM_UNIXTIME(creation_date))` is the obvious formulation and is correct, but wrapping an indexed column in a function prevents the database from using the index on it, forcing a full table scan. Comparing the raw timestamp against a computed range is equivalent and remains index-friendly. On the current data volume the difference is imperceptible; on several years of accumulated proposals it will not be, and the cost of writing it correctly now is one extra line.

8.3.4 Response generation

The CSV is streamed rather than assembled in memory:

```
1 public function submitForm(array &$form, FormStateInterface $form_state) {
2     $rows = $this->fetchRows(
3         $form_state->getValue('status'),
4         $form_state->getValue('year')
5     );
6
7     $response = new StreamedResponse(function () use ($rows) {
8         $handle = fopen('php://output', 'w');
9         fputcsv($handle, ['Name', 'Email', 'University', 'Lab Title',
10             'Status', 'Date of Submission']);
11         foreach ($rows as $row) {
12             fputcsv($handle, [
13                 $row->name,
14                 $row->email,
15                 $row->university,
16                 $row->lab_title,
17                 $this->statusLabel($row->approval_status),
18                 date('d-m-Y', $row->creation_date),
19             ]);
20         }
21     });
22 }
```

```

20     }
21     fclose($handle);
22 });
23
24 $filename = 'lab-migration-proposals-' . date('Y-m-d') . '.csv';
25 $response->headers->set('Content-Type', 'text/csv; charset=utf-8');
26 $response->headers->set('Content-Disposition',
27     'attachment; filename="' . $filename . '"');
28
29 $form_state->setResponse($response);
30 }

```

Listing 8.4: Streamed CSV response

Three details in this handler each address a specific failure mode.

StreamedResponse rather than a string. Building the entire file in a PHP string works until the result set is large enough to exhaust the memory limit, at which point the export fails with a fatal error and no partial output. Streaming writes each row as it is produced, so memory use is independent of row count.

fputcsv() rather than joining with commas. Proposal records contain lab titles and university names with commas, quotation marks and occasionally line breaks. Naïve concatenation corrupts the file structure at the first such value — the row acquires an extra column and the spreadsheet misaligns everything after it. `fputcsv()` applies the quoting and escaping rules of the format.

Status codes rendered as labels. The database stores approval status as an integer. Exporting the integer would require the recipient to know the encoding. The export writes “Approved” and “Pending”, using the same mapping the administrative screens use, so that the file is intelligible on its own.

setResponse() rather than a redirect. Drupal’s form workflow expects a submit handler to redirect. Setting a response object on the form state tells the form system to return that response directly, which is how a form can produce a file download instead of a page.

8.4 Delivered Interface

Figure 8.3 shows the export screen with both filters applied — status *Approved*, year *2026* — together with the matching records displayed for confirmation before download.

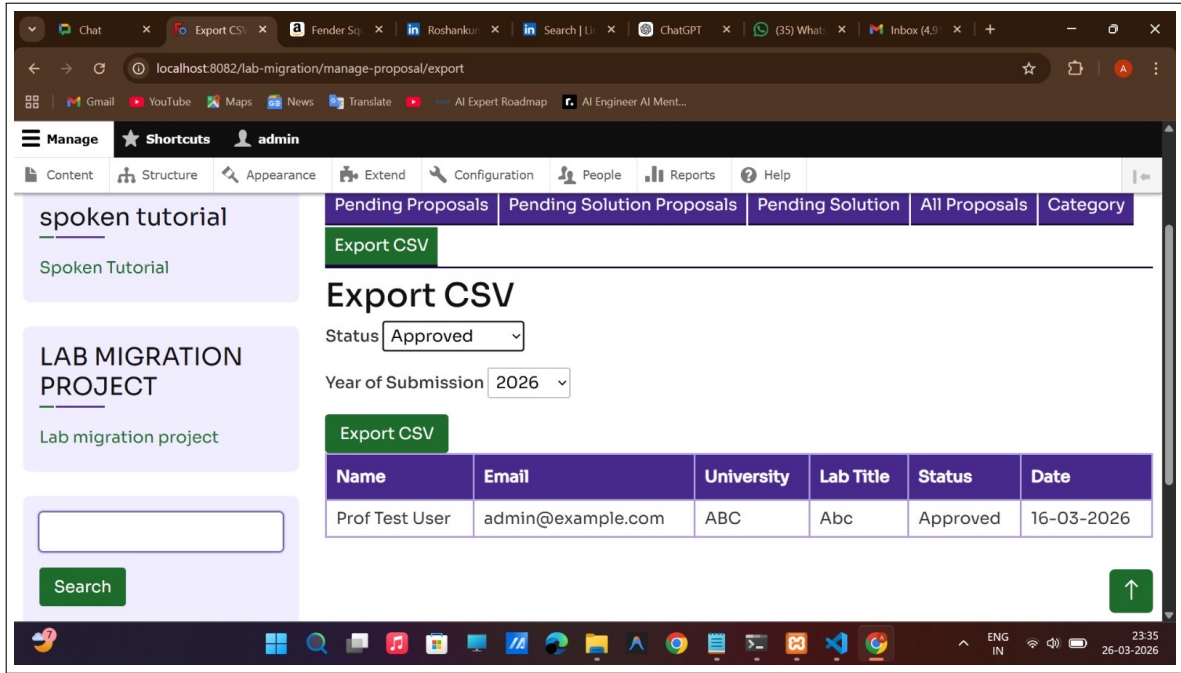


Figure 8.3: Export CSV with filters applied: approved proposals from 2026, with the matching records shown for confirmation

The confirmation table was not part of the original requirement. It was added after using the feature: with both filters set it is easy to produce an empty export by mistake — selecting a status and a year that have no overlap — and the download of an empty file is indistinguishable from a broken feature. Showing what will be exported makes an empty result obviously an empty result.

8.5 Reuse Across Modules

The same feature was implemented in the DWSIM Custom Model module against its proposal table. The second implementation was substantially faster than the first, because the design questions — where the route belongs, which permission applies, how filters compose, how the response is produced — had already been answered. What differs between the two is the table name, the column list and the status vocabulary; the structure is identical.

Table 8.1: Export CSV: exported fields

Column	Source	Transformation
Name	p.name	None
Email	p.email	None
University	p.university	None
Lab Title	p.lab_title	None
Status	p.approval_status	Integer mapped to label
Date of Submission	p.creation_date	Unix timestamp formatted d-m-Y

Chapter 9

Module-wise Implementation

The preceding chapters described the migration techniques and the two reusable services as general concerns. This chapter treats each module in turn: what it does, how it is structured, what its migration specifically involved, what went wrong, and what state it is in now.

9.1 FreeCAD Lab Migration

9.1.1 Purpose

The Lab Migration module implements the FreeCAD lab migration programme. An institution proposes to migrate a laboratory course from a proprietary CAD tool to FreeCAD; the proposal lists the lab, its department and its constituent experiments. Once approved, the proposer — or an assigned solution provider — uploads models and supporting files for each experiment. Reviewers approve these submissions, individually or in bulk, and completed labs are published on the public site. The module additionally issues participation and completion certificates as generated PDFs.

9.1.2 Architecture

Table 9.1: FreeCAD Lab Migration: structure

Element	Detail
Machine name	<code>lab_migration</code>
Form classes	20, covering proposal, solution proposal, approval, edit, status, bulk approval, code upload, certificates, settings and PDF generation
Controller	<code>DefaultController</code> , with the listing and detail pages
Services	Global helper functions and mail functions, plus the two services introduced during this work
Routes	42
Storage	Custom tables: <code>lab_migration_proposal</code> , <code>lab_migration_experiment</code> , <code>lab_migration_solution</code> , <code>lab_migration_solution_files</code> , <code>lab_migration_category</code> and others

9.1.3 Migration work

This module received the fullest treatment, since it was the first and served as the pattern for the rest:

- **Metadata.** `core_version_requirement` widened to `^10 || ^11`.
- **Mail.** All 11 `drupal_mail()` call sites migrated to the mail service; 15 `language_default()` calls replaced with the language manager.
- **Path arguments.** 32 `arg()` occurrences resolved — live calls converted to route parameters, commented remnants deleted.
- **AJAX.** Cascading-dropdown query logic extracted into `AjaxHelper`; 11 callback methods reduced to non-duplicated one-line delegates with standards-compliant names.
- **Dependency injection.** Every form touched during the work converted from static service lookups to constructor injection.
- **Link generation.** `\Drupal::l()` in the controller replaced with `Link::fromTextAndUrl()` and route-name based `Url` objects.
- **New feature.** Export CSV, described in Chapter 8.
- **Cleanup.** Commented Drupal 6 and 7 code, leftover `var_dump();die;` statements, and a `use` statement importing a class that has never existed (`Drupal\Core\LabMigration\PluginMail`).

9.1.4 Challenges

The mail service class was not a service. The class registered as `lab_migration_email` took no constructor arguments, and obtained a database connection by calling a

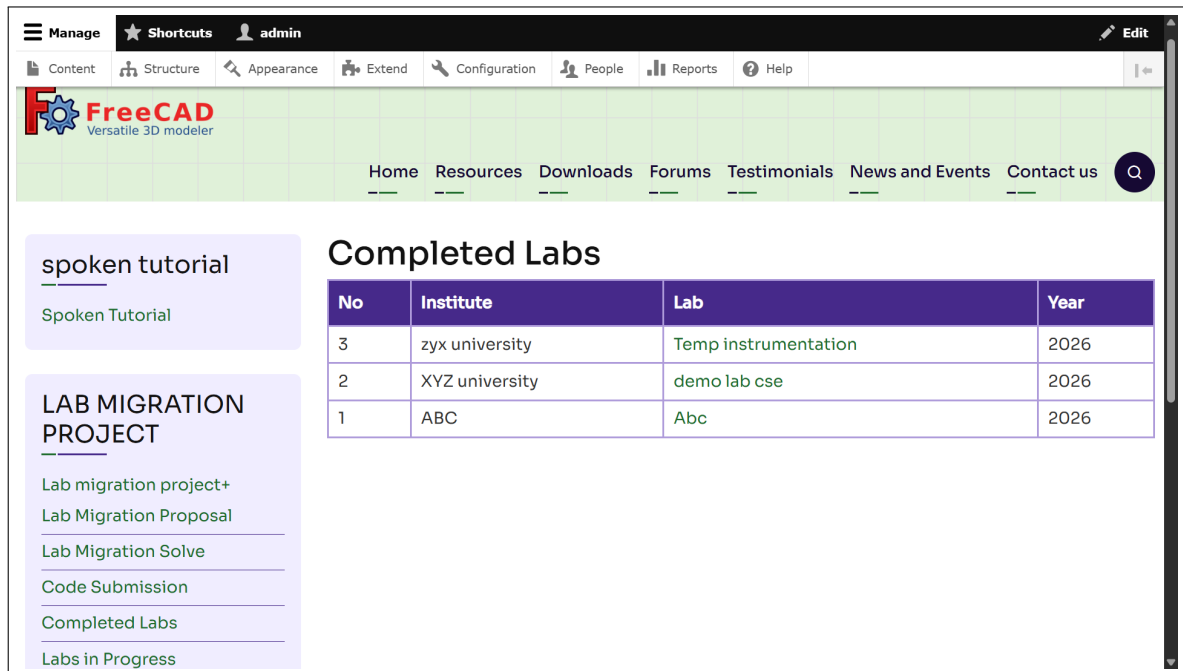
FormBase helper method that does not exist on an arbitrary class. Because the branch containing it was only reached when a particular notification fired, the error had never surfaced during ordinary use. Rebuilding the class around injected dependencies was a precondition for the mail work rather than a side effect of it.

Certificate forms reachable from multiple routes. The certificate and participation-certificate forms are reached through several paths that differ in prefix, and they used `arg()` with different indices in different branches to work out which record and which certificate type applied. Converting them required harmonising the routes so that a single named parameter carried the record identifier and the certificate type became an explicit route default rather than something inferred from path depth.

PDF generation. The certificate PDFs are produced by bundled third-party libraries invoked from module code. These were confirmed to run under PHP 8.3 and were left otherwise untouched; replacing them was outside the scope of the internship and would have introduced risk disproportionate to the benefit.

9.1.5 Results

The module installs and operates on Drupal 11. Every workflow step — proposal submission, solution proposal, approval, code upload, bulk approval, completion and certificate generation — was exercised and matched the Drupal 10 behaviour. Figure 9.1 shows the public listing of completed labs after the migration.



No	Institute	Lab	Year
3	zyx university	Temp instrumentation	2026
2	XYZ university	demo lab cse	2026
1	ABC	Abc	2026

Figure 9.1: Completed Labs listing on the migrated FreeCAD site

9.2 DWSIM Custom Model

9.2.1 Purpose

The Custom Model module supports contributions of custom thermodynamic property packages and unit-operation models to DWSIM. A contributor proposes a model with an abstract; a reviewer approves or rejects the abstract; on approval the contributor submits the implementation; submissions are reviewed and approved in bulk; and the entry is marked complete.

9.2.2 Workflow

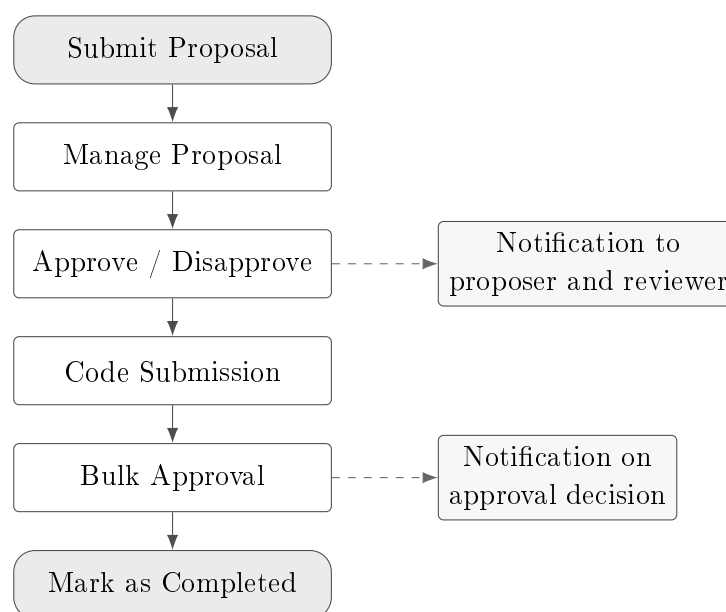


Figure 9.2: DWSIM Custom Model workflow

9.2.3 Migration work

The module comprises 12 form classes, one controller, two services and 27 routes. The work applied here was the pattern established on the FreeCAD module:

- Core version constraint widened to admit Drupal 11.
- 7 `drupal_mail()` call sites and 4 `language_default()` calls migrated to the mail service.
- 24 `arg()` occurrences resolved to route parameters or removed.
- AJAX option-building extracted into the module's helper service.
- Export CSV implemented against the custom model proposal table.
- Static service calls in touched classes replaced with injection.

9.2.4 Challenges

Status encoding differs from FreeCAD. Both modules store approval status as an integer, but the codes do not correspond: what is approved in one is not necessarily approved in the other, and the custom model workflow has an abstract-approval state with no FreeCAD analogue. This mattered when implementing the CSV export, where the integer is rendered as a label. The mapping was established by reading the submit handlers that write the column rather than by analogy with the FreeCAD module, and the export was verified against records in each state.

Bulk approval semantics. The bulk approval screen offers three actions: approve, mark as pending, and disapprove or delete. The delete path removes uploaded files as well as database records, and it is irreversible. Testing it required a database snapshot before each attempt, since a mistake could not be undone by re-running the workflow.

9.2.5 Results

The module runs on Drupal 11 with the reusable mail and AJAX services in place and an Export CSV facility matching the FreeCAD one. The complete workflow — submit, manage, approve, submit code, bulk-approve, complete — was exercised end to end.

9.3 DWSIM Flowsheet

9.3.1 Purpose

The Flowsheet module supports the DWSIM flowsheeting project, in which contributors reproduce process flow diagrams as DWSIM simulations. Its workflow is shorter than the Custom Model one — proposal, approval, flowsheet upload, approval, publication — but it has the heaviest AJAX usage of the four modules, with 15 elements carrying AJAX behaviour.

9.3.2 Migration work

- Core version constraint widened.
- 3 `drupal_mail()` call sites and 3 `language_default()` calls migrated to the mail service.
- 22 `arg()` occurrences resolved.
- `drupal_set_message()` calls replaced with the messenger service.
- AJAX consolidation, which had the largest proportional effect here given the density of AJAX-bearing elements.
- Approval workflow verified against the Drupal 10 behaviour, including the notification sent on each transition.

9.3.3 Challenges

AJAX and #states interacting. Several elements carried both an '`#ajax`' attachment and a '`#states`' visibility condition. The two mechanisms operate at different layers — `#states` is evaluated in the browser, AJAX rebuilds the element server-side — and an element that is rebuilt by AJAX loses its client-side state bindings unless the wrapper is structured so that the states are reattached with the replaced markup. Two screens exhibited controls that became permanently hidden after an AJAX rebuild. The fix was to place the wrapper outside the element carrying the states condition rather than around it.

Sparse notification coverage. The module sent only three notifications, noticeably fewer than the equivalent points in the other modules. Whether this was intentional or an omission could not be determined from the code or from the commit history, so the existing behaviour was preserved exactly and the observation was raised with the mentor rather than resolved unilaterally. Adding a notification is a product decision, not a migration decision.

9.3.4 Results

The module runs on Drupal 11, dispatches its notifications through the shared mail service, and its AJAX-driven screens behave as before.

9.4 DWSIM Textbook Companion

9.4.1 Purpose

The Textbook Companion module is the largest of the four. Contributors reproduce solved examples from standard chemical engineering textbooks as DWSIM simulations. The data model is correspondingly deeper — books contain chapters, chapters contain examples, examples have code submissions — and the workflow includes a code review stage in which a reviewer examines the submitted simulation and either approves it or returns it with comments.

9.4.2 Structure

With 16 form classes and 48 routes, this module has the widest surface of the four. It also had, by a considerable margin, the heaviest legacy load: 44 `drupal_mail()` calls, 44 `language_default()` calls, 77 `arg()` occurrences and 486 `drupal_set_message()` calls.

9.4.3 Migration work

- **Forms.** All 16 form classes reviewed; those containing legacy API calls migrated and converted to constructor injection.

- **Controllers.** Listing and detail pages migrated, including link generation and path-argument access.
- **Services.** The module's helper services rebuilt around injected dependencies.
- **Routing.** All 48 routes reviewed; parameters named consistently so that forms could receive them by name.
- **Mail.** The reusable mail service introduced, eliminating 44 calls to the removed API.
- **Messenger.** 486 `drupal_set_message()` calls replaced, each mapped to `addStatus()`, `addWarning()` or `addError()` according to its severity argument.
- **AJAX.** Duplicated callbacks in the book, chapter and example forms consolidated.

9.4.4 Challenges

Volume. This module is where the strategy of establishing patterns elsewhere first paid for itself. There is no clever way to migrate 44 mail call sites; there is only a way to make each one take two minutes instead of twenty, which is to have already decided what the target shape is. Working through them mechanically, in small commits, was the entire technique.

Messenger severity mapping. The 486 message calls could not be transformed by a single search and replace, because the severity is encoded in an optional second argument that is absent for status messages, present as `'warning'` or `'error'` otherwise, and occasionally passed as a variable. Calls with a literal severity were transformed by pattern; calls with a variable severity were converted by hand after reading the surrounding branch.

Depth of the data model. The three-level hierarchy means that AJAX cascades are three deep and that deletion has to consider descendants. A chapter deleted without its examples leaves orphaned rows that appear in listings as entries with no parent. The deletion paths were checked against this specifically, since a migration that changes the order of operations in a delete handler can introduce exactly this class of defect.

9.4.5 Results

The module runs on Drupal 11. All forms, controllers and services were migrated, routing was updated across all 48 routes, mail is dispatched through the reusable service, and duplicated AJAX callbacks were removed.

9.5 Consolidated View

Table 9.2: Work performed per module

Module			D11	Mail	AJAX	Args	DI	CSV	Cleanup
FreeCAD	Lab	Migra- tion	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DWSIM	Custom Model		Yes	Yes	Yes	Yes	Yes	Yes	Yes
DWSIM	Flowsheet		Yes	Yes	Yes	Yes	Yes	—	Yes
DWSIM	Textbook		Yes	Yes	Yes	Yes	Yes	—	Yes
Companion									

Chapter 10

Application Workflow

10.1 The Common Workflow

All four modules implement variations of a single process. Understanding that process as a whole is what made the shared services possible: components were extracted because the same step occurs in every module, not because the code happened to look similar.

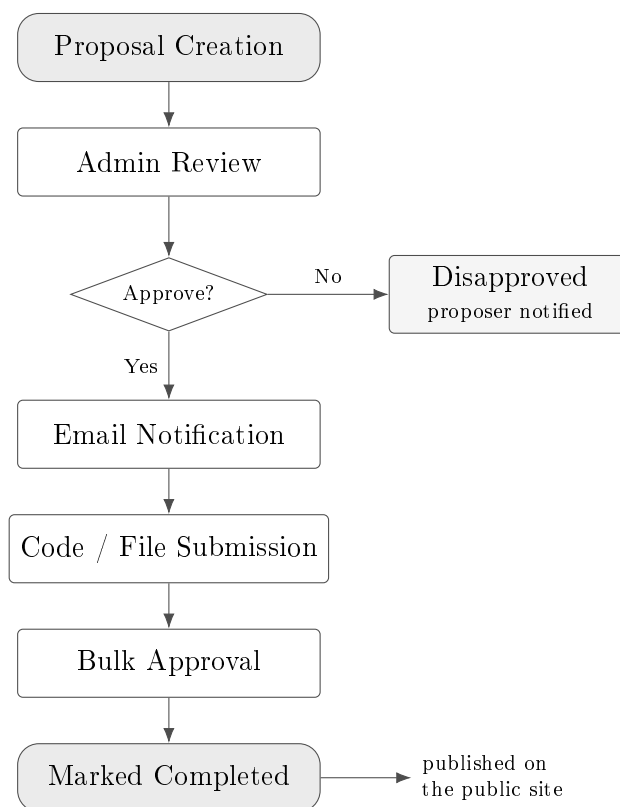


Figure 10.1: End-to-end workflow common to the FOSSEE Drupal applications

10.2 Stage by Stage

10.2.1 Proposal creation

A registered contributor completes the proposal form. The form collects identifying details — name, institution, department, contact address — and the substance of the proposal: the lab and its experiments, the model and its abstract, the flowsheet, or the book and its chapters, depending on the module. Cascading dropdowns are used wherever a selection constrains a later one, and these are the elements now served by the AJAX helper.

On submission the record is written with a pending status and a notification is dispatched — to the proposer as an acknowledgement, and to the reviewing team as a work item. Both dispatches now go through the mail service.

10.2.2 Administrative review

Reviewers work from listing screens that partition proposals by state: pending proposals, pending solution proposals, pending solutions, and all proposals. These are controller-rendered tables with per-row actions. Figure 8.1 in Chapter 8 shows the All Proposals screen, including the Export CSV tab added during this work.

10.2.3 Approval decision

The approval form records the decision, writes the new status, and dispatches the corresponding notification. Approval opens the submission stage; disapproval closes the record with the reason communicated to the proposer.

This is the point in the workflow where the correctness of the mail migration is most consequential. An approval that is recorded in the database but whose notification silently fails leaves the proposer with no indication that they may proceed — the application shows the state correctly, but nobody looks at the application until they have been told to.

10.2.4 Code and file submission

The contributor uploads the artefacts: FreeCAD models and supporting files, DWSIM simulation files, custom model implementations, or example code. Files are stored under the site's file directory with database rows linking them to the parent record. Upload forms use the same cascading pattern — select the parent record, then the sub-record, then attach files — and are therefore also served by the AJAX helper.

10.2.5 Bulk approval

Reviewing submissions individually does not scale: a single lab migration proposal may carry dozens of experiments, each with its own solution files. The bulk approval screen presents all pending submissions for a selected parent record and applies a decision to many at once — approve, return to pending, or disapprove and delete.

The delete action is the most destructive operation in the application. It removes uploaded files as well as database rows and cannot be undone from the interface. This is why testing it required database snapshots rather than repeated workflow runs.

10.2.6 Completion

When all submissions for a record are approved, the record is marked complete. It then appears on the public-facing listings — completed labs, published flowsheets, available custom models, completed textbook companion entries — where it becomes available for download by anyone. Figure 9.1 in Chapter 9 shows this listing on the migrated site.

10.3 Administrative Workflow

Alongside the contribution workflow, the modules provide administrative facilities that are not part of any single proposal's lifecycle:

- **Category management** — maintaining the taxonomy against which proposals are classified.
- **Settings** — module configuration, including the sender address used for notifications. After the mail centralization, this single value governs every outbound message from the module rather than being read independently in each form.
- **Certificates** — issuing participation and completion certificates as generated PDFs.
- **Export CSV** — extracting proposal data for reporting, described in Chapter 8.

10.4 Notification Map

Table 10.1 lists the workflow transitions that dispatch mail. It was compiled during the migration as a checklist: each row was verified to fire after the change as it had before, since a notification silently lost during refactoring is invisible until a user reports not having received it.

Table 10.1: Workflow transitions that dispatch notifications

Transition	Recipient	Purpose
Proposal submitted	Proposer	Acknowledgement with submitted details
Proposal submitted	Reviewing team	New item awaiting review
Proposal approved	Proposer	Permission to proceed to submission
Proposal disapproved	Proposer	Decision and reason
Solution proposal approved	Proposer	Assignment confirmed
Code or files submitted	Reviewing team	Submission awaiting approval
Submission approved	Contributor	Confirmation
Submission deleted	Contributor	Notice that the submission was removed
Record completed	Contributor	Completion and publication notice

Chapter 11

Testing and Validation

11.1 Testing Strategy

The modules carry no automated test suite, and building one was outside the scope of the internship. Verification was therefore manual and workflow-driven, and it was organised around a principle that follows directly from the nature of the work: because the objective was to preserve behaviour rather than to add it, the test oracle is the existing system.

Every check took the form of a comparison. The Drupal 10 site was kept running alongside the Drupal 11 one, populated with equivalent test data, so that any uncertainty about how a screen ought to behave could be settled by looking rather than by reasoning about the code.

Four levels of testing were applied:

1. **Installation testing** — does the module install and enable on a Drupal 11 site without errors?
2. **Screen testing** — does each route render without a fatal error or a log entry?
3. **Workflow testing** — does a record traverse the complete lifecycle and reach the same end state?
4. **Regression testing** — do previously working screens still work after a change made elsewhere?

11.2 Test Matrix

Table 11.1: Workflow test matrix

Test case	Procedure	Expected result
Proposal creation	Submit the proposal form as a registered contributor	Record written with pending status; acknowledgement and reviewer notifications dispatched
Validation	Submit with mandatory fields blank and with malformed input	Field-level errors displayed; no record written
Cascading dropdowns	Select parent, verify child options; change parent, verify child resets	Options match the parent selection; no stale values retained
Proposal approval	Approve from the management screen	Status updated; approval notification logged and dispatched
Proposal disapproval	Disapprove with a reason	Status updated; disapproval notification carries the reason
Code submission	Upload files against an approved record	Files stored; database rows linked; reviewer notified
Bulk approval	Approve several submissions in one operation	All selected records updated; one notification per contributor
Bulk disapproval and delete	Delete submissions in bulk (from a database snapshot)	Rows and files removed; no orphaned records in listings
Completion	Mark a record complete	Record appears in the public completed listing
Export CSV, unfiltered	Submit with both filters at <i>Any</i>	File contains every proposal; header row correct
Export CSV, filtered	Select a status and a year	File contains only matching records; on-screen preview agrees
Export CSV, empty result	Select a status and year combination with no records	Header-only file; empty preview shown rather than a silent download
Export CSV, special characters	Export records whose titles contain commas and quotation marks	Columns remain aligned when the file is opened in a spreadsheet
Certificate generation	Generate a participation certificate	PDF produced with correct details

11.3 Mail Verification

Mail is the hardest part of the workflow to verify, because success in the application does not imply delivery. Three complementary checks were used.

Log inspection. The mail service logs every attempt with its key and recipient, so `drush ws` shows what the application believes it sent. This distinguishes a notification that was never triggered from one that was triggered and rejected.

Local capture. In development, mail was captured rather than delivered, so that message bodies could be inspected without sending anything to real contributors. This was essential: several notifications embed record details in the body, and the only way to confirm that the parameter array reaches `hook_mail()` in the expected structure is to read the rendered message.

Return value verification. The specific defect described in Chapter 7 — treating the mail manager’s array return as a boolean — was verified fixed by deliberately configuring an invalid mail transport and confirming that a failure appeared in the log rather than passing unnoticed.

11.4 Regression Testing

Regression was the real risk. The migration touched files that participate in several workflows, so a change made for one screen can break another that was never opened.

Two practices addressed this. First, a short regression pass — submit, approve, upload, bulk-approve, complete — was run after every significant change rather than only at the end of a module. Second, when a defect was found, the workflow step immediately before and after the failing one was re-tested as well, on the assumption that a wrong value written in one step usually surfaces in the next.

The defects found by regression testing during the internship clustered into three categories, each of which is a general lesson rather than an incident:

- **Lost notifications.** A refactored submit handler that no longer dispatched mail because the call had been moved inside a conditional branch that was not always taken.
- **Status value drift.** A status written as a different integer after a handler was restructured, which left the record in a state no listing screen queried for — it simply vanished from the interface while remaining in the database.
- **Stale AJAX selections.** A dependent dropdown retaining the previous parent's options after the parent changed, which was traced to the wrapper element being rebuilt while the child's default value was not cleared.

11.5 Browser and Environment Testing

Administrative screens were checked in Chrome and Firefox. The AJAX behaviour was the specific concern, since cascading dropdowns depend on JavaScript, and the screens were verified at both desktop and narrow viewport widths — the administration theme reflows tables at narrow widths and the action links in the final column are the first thing to become unreachable.

The site was also exercised on both container runtimes used during the internship, Docker and Podman, to confirm that nothing in the module behaviour depended on the environment. The only difference observed was in file upload handling, which traces to the file ownership and permission differences described in Chapter 4 rather than to the modules themselves.

11.6 Cache Rebuild Discipline

A procedural point that deserves recording, because it caused more confusion than any genuine defect: many apparent failures during testing were stale caches.

Drupal caches routing, service definitions, plugin definitions and hook implementations. A new route returns “page not found”; a new service throws `ServiceNotFoundException`; a new `hook_mail()` implementation is not invoked. All

three look exactly like code errors.

The practice adopted was to run `drush cr` before investigating anything. It costs a few seconds and eliminates an entire class of false diagnosis. Several hours were lost to this in the first two weeks and effectively none afterwards.

11.7 Limitations of the Testing Performed

Manual testing is not a substitute for an automated suite, and it is worth being precise about what it does not provide.

It does not provide repeatability: the confidence that the workflow was correct on the day it was tested does not extend to a change made a month later by someone else. It does not provide coverage measurement, so there is no way to know which branches were never exercised. And it does not scale — the regression pass takes several minutes per module, which is affordable a few times a day and not affordable on every commit.

This is why an automated functional test suite is the first item in the future scope (Chapter 15). The workflows are now stable and well-understood, and the shared services have made the logic accessible outside a rendered form. Both conditions make the modules considerably easier to write tests for than they were at the start of the internship.

Chapter 12

Challenges Faced

This chapter records the difficulties encountered and how each was resolved. The intent is practical: most of these are problems that the next person to work on these modules will meet, and the resolution is more useful than the description.

Table 12.1: Summary of challenges and resolutions

Challenge	Nature	Resolution
Composer dependency conflicts	Contributed modules constraining core to ~10	<code>composer why-not</code> to identify blockers; update or remove
Container setup	Missing PHP extensions; file ownership across bind mounts	Extensions added at image build; container UID aligned with host UID
Stale caches	Routes, services and hooks not taking effect	<code>drush cr</code> as a reflex before any diagnosis
Silent mail failure	Array return treated as boolean	Check <code>\$message['result']</code> ; log every attempt
<code>arg()</code> conversion	Requires knowing the route, not a textual substitution	Route parameters named consistently; routes harmonised
Deprecation tools miss old APIs	Drupal 7 functions predate the deprecation database	Direct source search alongside Upgrade Status
Duplicated logic diverged	Two copies disagreeing on filtering	Difference made an explicit parameter, not resolved silently
AJAX and <code>#states</code> interaction	Rebuilt elements losing client-side state bindings	Wrapper placed outside the states-bearing element
Destructive bulk operations	Delete removes files and rows irreversibly	Database snapshot before each test run
Codebase volume	486 messenger calls, 65 mail calls, 155 <code>arg()</code> occurrences	Patterns settled on a small module first, then applied mechanically

12.1 Composer and the Dependency Graph

The first substantial obstacle was that the core upgrade would not resolve. Composer's output on failure lists constraints from across the dependency graph without identifying which one is responsible, and the instinct — to relax constraints until something works — produces a dependency set that installs and is not defensible.

`composer why-not drupal/core-recommended 11` answers the question directly, naming the packages that block the upgrade. Each blocker then reduces to a decision: update it if a compatible release exists, or remove it if the module is not genuinely required. Pinning a blocker to an old version was rejected as an option, since carrying

forward an unmaintained dependency defeats the purpose of upgrading.

12.2 Container Environment

Two failures dominated the environment setup.

The stock PHP image lacks `pdo_mysql`, `gd` and `zip`. Drupal's installer reports these at its final step, after database configuration has already succeeded, which makes the failure appear more serious than it is. Installing the extensions during the image build rather than patching a running container was the difference between an environment that can be recreated and one that exists only on one machine.

File ownership across the bind mount was subtler. Under Docker the web server writes as a container user with no relation to the host user, so Drupal cannot create its settings file and uploads fail without a clear message. Aligning the container user's UID with the host user's UID at build time resolved it. Podman, running rootless, does not have this problem but introduces an SELinux labelling requirement on the mount instead. Both configurations were documented in the repository rather than left as knowledge held by one person.

12.3 Working Without Reliable Error Messages

Drupal's failure modes are frequently uninformative at the point of failure. A fatal error in a module produces a white screen; the actual exception is in the watchdog log or the PHP error log. A service that cannot be constructed fails when the form is instantiated, which is a different place from where the service is used. A missing route presents as a 404 that looks like a bad link.

The habits that made this manageable were, in order of value: rebuild caches before diagnosing; read `drush ws` before reading the code; and enable verbose error display in the development environment so that the exception appears in the browser rather than only in a log.

12.4 Legacy Code Whose Intent Is Unrecoverable

The most genuinely difficult problems had nothing to do with Drupal.

Two copies of the same option builder disagreed about whether to filter by approval status. The correct behaviour depends on what the screen is for, and nothing in the code, the comments or the commit history records which copy was right. Choosing one and applying it to both would have changed a screen's behaviour silently — exactly the outcome the migration was supposed to avoid. The resolution was to preserve both behaviours behind an explicit parameter, so that the difference is now visible and can be settled by whoever knows the answer.

Similarly, the DWSIM Flowsheet module sends notifications at only three points

where the other modules send them at eight or nine. This may be deliberate or it may be an omission. The existing behaviour was preserved and the observation was raised with the mentor. Adding a notification changes what users receive, which is a product decision that a migration is not entitled to make on its own.

12.5 The Discipline of Scope

The persistent temptation across five months was to fix things that were not broken in the relevant sense. A form with a thousand-line `buildForm()` method, a controller returning HTML assembled by string concatenation, a database schema without foreign keys — all of these are real weaknesses, and none of them prevents the module from running on Drupal 11.

The rule that was applied, and that took some discipline to hold, was to fix what the objective required plus what the code being touched anyway made cheap, and to record everything else as future scope. The alternative — an internship that produced a partially rewritten codebase in which nothing was finished — would have been worth considerably less than one that produced four migrated modules and a written account of what remains.

Chapter 13

Results

13.1 Migration Outcomes

All four modules were migrated to Drupal 11 and verified against their Drupal 10 behaviour.

Table 13.1: Migration status by module

Module	Installs on D11	Workflow verified	Additional work delivered
FreeCAD Lab Migration	Yes	Yes	Mail service, AJAX helper, Export CSV
DWSIM Custom Model	Yes	Yes	Mail service, AJAX helper, Export CSV
DWSIM Flowsheet	Yes	Yes	Mail service, AJAX consolidation
DWSIM Textbook Companion	Yes	Yes	Mail service, AJAX consolidation, forms and controllers migrated

13.2 Legacy API Elimination

Table 13.2: Legacy API occurrences before and after

API	Before	After	Status
drupal_mail()	65	0	Removed
language_default()	66	0	Removed
arg()	155	0	Removed
drupal_set_message()	488	0	Removed
\Drupal::l()	2	0	Removed
Commented db_query() blocks	Numerous	0	Removed

The before-counts include occurrences inside commented-out code, which was removed rather than migrated. The after-counts are zero because a removed API in a comment is as much a liability as one in live code: it documents an approach that cannot work and invites restoration.

13.3 Reduction in Duplication

Table 13.3: Duplication before and after consolidation

Measure	Before	After
Files containing mail dispatch logic	21	4
Places reading the sender configuration	21	4
Duplicated AJAX callback bodies	11	0
Places querying experiment options	4	1
Places querying solution options	3	1
Modules able to reuse another module's mail routine	0	All

The four remaining mail-dispatch files are one service per module. Consolidating further — a single service shared by all four — was considered and rejected: it would create a dependency between modules that are separately installable, and each module's configuration, sender address and message keys are genuinely its own.

13.4 Functional Additions

The Export CSV feature was delivered for the FreeCAD Lab Migration and DWSIM Custom Model modules, with filtering by approval status and year of submission, CSV-correct escaping of values containing commas and quotation marks, an on-screen preview of what will be exported, and streamed output that does not scale its memory use with the size of the result set.

Beyond the feature itself, the mail service introduced per-attempt logging, which gives the administrative team a way to answer a question that previously had no answer: whether a notification a contributor says they never received was in fact dispatched.

13.5 Qualitative Outcomes

Some of the more valuable results do not reduce to a table.

Failures are now visible. Mail failures were previously either unreported or reported to a screen nobody was looking at. Every dispatch is now logged, and failures are logged as errors.

Divergences are now explicit. Behavioural differences that were previously hidden in duplicated method bodies — one copy filtering by approval status, another not — are now parameters at the call site, visible to anyone reading the code.

Dependencies are now declared. Classes touched during the work state their collaborators in their constructors instead of fetching them from a global registry, which is the precondition for the automated testing that the modules still lack.

The codebase is smaller. Removing commented legacy code, debugging statements, unused imports and unreachable functions did not change behaviour at all, and it removed a substantial amount of material that a future maintainer would otherwise have had to read and evaluate.

13.6 Deliverables

Table 13.4: Deliverables

Deliverable	Detail
Migrated modules	Four custom modules running on Drupal 11
Reusable mail service	Per module, with logging and cross-module invocation
Reusable AJAX helper	Per module, replacing duplicated option builders
Export CSV	Two modules, with status and year filtering
Deprecated API inventory	Audit record used to drive the migration
Environment documentation	Docker and Podman setup notes in the repositories
Version control record	All work on the <code>sli-intern</code> branch, reviewed

Chapter 14

Learning Outcomes

14.1 Technical

Drupal architecture

Working through four modules gave a working understanding of Drupal as an application framework rather than as a content management system: the request lifecycle from routing through controller resolution to the render pipeline, the Form API's build-validate-submit cycle, the plugin system as it applies to mail, and the database abstraction layer's query builder.

The most useful realisation was about layering. These applications store almost nothing as Drupal entities; they use custom tables and treat Drupal as infrastructure. That is a legitimate architecture, and recognising it early explained why the migration concentrated on forms, routing and services and barely touched the entity and field APIs where most Drupal upgrade documentation focuses.

Dependency injection and the service container

The distinction between service location and dependency injection was, before this internship, something understood as a definition. Working in a codebase with over a thousand static service calls made the practical difference concrete: a class that fetches its collaborators cannot be read to determine what it needs, cannot be tested without a bootstrapped container, and cannot have a collaborator substituted without editing its internals.

The cross-module mail requirement demonstrated the container's real value. Two modules that share no class hierarchy can share a component addressed by name, with the failure mode — a missing service — surfacing loudly at construction rather than quietly at use.

Managing deprecation

Drupal’s deprecation policy is a specific instance of a general problem: how a platform evolves without breaking everything built on it. Seeing the consequence of deferring that work — sixty-five calls to a function removed several major versions ago, each a latent fatal error — was more instructive than any description of the policy.

The related lesson concerns tooling. Upgrade Status and Rector are valuable and incomplete. They detect what they have rules for, and the oldest problems in a long-lived codebase are precisely the ones that have fallen out of the deprecation database. A tool-only audit would have declared these modules nearly clean.

Composer and dependency management

Resolving the core upgrade taught the practical use of Composer beyond `require` and `install`: reading a constraint failure, using `why-not` to identify the blocking package, and understanding why the lock file matters. It also settled a judgement question — pinning a blocker to an old version makes the immediate error disappear and makes the underlying problem worse.

Refactoring under a behaviour-preservation constraint

Extracting two services from duplicated code, while guaranteeing that no screen changed, was the most substantial design work of the internship. The techniques that made it tractable: preserve the value contract so that call sites can be converted one at a time; make discovered divergences explicit parameters rather than resolving them silently; and separate the extraction commit from any behavioural commit so that a reviewer can verify each independently.

14.2 Software Engineering Practice

Version control and code review

Working on a review-gated branch changed how commits were composed. A commit is not a save point; it is a unit of explanation. One logical change per commit is what makes review possible, and review is where the most valuable feedback of the internship arrived — including the suggestion to extract shared services, which redefined the project.

Reading code written by others

More time was spent reading than writing. The specific skill developed was reconstructing intent from code that carries none: determining what a status integer means by finding the handler that writes it, working out which route reaches a form by searching the routing file for its class name, and distinguishing a deliberate difference between

two similar functions from an accidental one.

The corollary was learning when reconstruction fails and the honest answer is to preserve existing behaviour and raise the question, rather than to guess.

Debugging production-shaped systems

Debugging a system whose errors appear somewhere other than where they occur required a different approach from debugging a program that prints a stack trace. The working method became: eliminate caching as a cause first, consult the log before the source, reproduce the failure at the smallest scope that still exhibits it, and change one thing at a time.

Scope discipline

The clearest professional lesson was about restraint. A legacy codebase presents an unlimited number of things worth improving, and an internship is five months long. Distinguishing what the objective requires from what is merely desirable — and writing the latter down as future scope instead of half-doing it — is what made four finished modules possible rather than several unfinished ones.

14.3 Open Source Development

Contributing to a real open-source project made concrete a set of expectations that are easy to state and harder to internalise: that changes are public and attributed; that a commit message is written for a reader who lacks all context; that a fix and a refactor belong in separate commits; and that code which works but cannot be reviewed has not yet been contributed.

Equally, working inside a project maintained by a rotating set of student developers gave a direct view of what happens to a codebase when each contributor optimises for their own immediate task. Almost every difficulty described in Chapter 12 originated in a reasonable local decision — copy the callback rather than refactor, keep the old query as a comment, use the static service call because it is shorter — whose cost was paid by someone else later. That is a more persuasive argument for engineering discipline than any style guide.

Chapter 15

Conclusion and Future Scope

15.1 Conclusion

The internship set out to migrate the FOSSEE FreeCAD and DWSIM custom Drupal modules from Drupal 10 to Drupal 11 and to modernize their internal structure along the way. Both objectives were met across all four modules in scope.

On the migration itself, the four modules — FreeCAD Lab Migration, DWSIM Custom Model, DWSIM Flowsheet and DWSIM Textbook Companion — install and operate on Drupal 11. Their metadata admits the new core version, and the legacy APIs they depended on have been replaced with supported equivalents: sixty-five calls to a mail function removed in Drupal 8, a hundred and fifty-five path-argument calls, and several hundred message and language calls. Every workflow was exercised end to end and matched its behaviour on the previous version, which was the binding constraint on the whole exercise.

On modernization, two reusable services were introduced. The mail service replaced per-form dispatch logic scattered across twenty-one files, corrected a defect that was silently suppressing mail failure reporting, and added per-attempt logging. The AJAX helper replaced duplicated callbacks and option builders with a single queryable component, and in doing so surfaced a behavioural divergence between two copies that had gone unnoticed — which was resolved by making the difference explicit rather than by choosing one silently. An Export CSV feature was designed and delivered for two modules, giving the administrative team a reporting capability it did not previously have.

What the work is worth beyond the immediate deliverables is best judged by the state the modules are left in. The obstacles that made the initial migration difficult — undeclared dependencies, duplicated logic, mail failures that reported nothing, dead code preserved in comments — are the same obstacles that would have confronted the next contributor. Removing them is the part of the work most likely to compound.

The technique that made the project tractable is worth stating once more, because it was not obvious at the outset and was arrived at by failing first. Auditing the source before attempting to run it inverts the economics of a legacy migration: failure sites in a Drupal application are often several workflow steps deep and appear as blank

pages rather than as errors, so discovering them by execution is slow and incomplete, while discovering them by search is fast and exhaustive. The audit-first approach found problems that a tool-driven scan alone would have missed entirely, because the oldest APIs in the codebase predate the deprecation infrastructure the tools rely on.

15.2 Future Scope

The following items were identified during the work and deliberately left undone. They are listed in the order in which they would most usefully be taken up.

Automated functional testing

This is the highest-value next step by a wide margin. Every claim of behavioural preservation in this report rests on manual testing, which is not repeatable, not measurable and does not scale to running on each commit. Drupal's functional test framework can drive the complete workflow — submit, approve, upload, bulk-approve, complete — programmatically. The dependency injection introduced during this work is what makes the shared services unit-testable in isolation, so the groundwork is in place.

Completing the dependency injection conversion

Classes not touched during this work retain their static service lookups. Over a thousand such calls remain. There is no urgency — they function correctly — but converting them incrementally, as each class is next modified, would eventually make the whole codebase testable rather than only the parts that have been revisited.

Content and data migration

Migrating production data onto the Drupal 11 sites was performed only to the extent needed for testing. A production cutover needs a rehearsed migration plan with verification counts per table and a rollback path.

Extending Export CSV

The export exists for two modules. The DWSIM Flowsheet and Textbook Companion modules would benefit from the same facility, and the implementation is now well-understood. A natural extension is a date-range filter in place of the single-year selector, and exporting the child records — experiments, examples — rather than only the parent proposal.

Refactoring oversized form classes

Several form classes exceed a thousand lines, with `buildForm()` methods that construct multiple screens' worth of elements behind conditionals. These work, and rewriting them was correctly out of scope, but they are the main remaining barrier to comprehension for a newcomer. Splitting them along the conditional boundaries that already exist would be a self-contained project.

Performance

The listing screens query without pagination and, in places, execute one query per row to fetch related records. This is imperceptible at the current data volume and will not remain so. Adding pagination and replacing per-row queries with joins is straightforward, and the year-filter implementation in the CSV export (Chapter 8) illustrates the general principle: keep indexed columns out of function calls in `WHERE` clauses.

Composer and dependency hygiene

Contributed modules should be reviewed periodically rather than at upgrade time. A module without a compatible release is a decision that has to be made under time pressure during an upgrade and can be made calmly at any other time.

Continuing the core upgrade cycle

The most important recommendation is procedural rather than technical. Drupal 11 will deprecate APIs during its cycle, and those deprecations will be removed in Drupal 12. Deprecation notices for exactly that code will be visible in the logs of the Drupal 11 site from now on. Acting on them continuously — rather than accumulating five years of them and then discovering, as this internship did, that a single removed function is called sixty-five times — is what would make the next migration a routine task instead of a semester-long project.

Chapter 16

Bibliography

1. FOSSEE, IIT Bombay. *FOSSEE Official Website*. [Online]. Available: <https://fossee.in>
2. FOSSEE, IIT Bombay. *FreeCAD Project*. [Online]. Available: <https://freecad.fossee.in>
3. FOSSEE, IIT Bombay. *DWSIM Project*. [Online]. Available: <https://dwsim.fossee.in>
4. Drupal Association. *Drupal 11 Release Notes*. [Online]. Available: <https://www.drupal.org/project/drupal/releases>
5. Drupal Association. *Upgrading from Drupal 10 to Drupal 11*. Drupal Documentation. [Online]. Available: <https://www.drupal.org/docs/upgrading-drupal>
6. Drupal Association. *Deprecated API Policy and Backwards Compatibility*. Drupal Core Documentation. [Online]. Available: <https://www.drupal.org/about/core/policies>
7. Drupal Association. *Services and Dependency Injection in Drupal*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/services-and-dependency-injection>
8. Drupal Association. *Form API*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/form-api>
9. Drupal Association. *AJAX API*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/ajax-api>
10. Drupal Association. *Mail API and `hook_mail()`*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/mail-api>
11. Drupal Association. *Routing System*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/routing-system>
12. Drupal Association. *Database API: Dynamic Queries*. Drupal API Documentation. [Online]. Available: <https://www.drupal.org/docs/drupal-apis/database-api>
13. Drupal Association. *PHP Coding Standards for Drupal*. [Online]. Available: <https://www.drupal.org/docs/develop/standards>

14. *Upgrade Status* contributed module. [Online]. Available: https://www.drupal.org/project/upgrade_status
15. *Drupal Rector*. [Online]. Available: <https://github.com/palantirnet/drupal-rector>
16. Symfony SAS. *Symfony 7 Documentation — HttpKernel, DependencyInjection and HttpFoundation Components*. [Online]. Available: <https://symfony.com/doc/current/index.html>
17. Composer. *Composer Documentation*. [Online]. Available: <https://getcomposer.org/doc/>
18. Drush. *Drush Command Line Documentation*. [Online]. Available: <https://www.drush.org/>
19. The PHP Group. *PHP 8.3 Manual — fputs, Streams and Output Buffering*. [Online]. Available: <https://www.php.net/manual/en/>
20. Docker Inc. *Docker Documentation*. [Online]. Available: <https://docs.docker.com/>
21. Podman. *Podman Documentation*. [Online]. Available: <https://docs.podman.io/>
22. M. Fowler. *Refactoring: Improving the Design of Existing Code*, 2nd ed. Addison-Wesley, 2018.
23. M. C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

Appendix A

Repositories and Version Control

A.1 Repositories Worked On

Table A.1: Repositories and their role in the internship

Repository	Branch used	Contents
freecad_lab_migration	sli-intern	FreeCAD Lab Migration custom module: forms, controller, services, routing, permissions, PDF and certificate generation
dwsim_custom_model	sli-intern	DWSIM Custom Model module: proposal, abstract approval, code submission and bulk approval workflow
dwsim_drupal10	sli-intern	DWSIM site repository containing the custom modules under <code>modules/custom/</code> , including DWSIM Flowsheet and DWSIM Textbook Companion
Local Drupal 11 site	main	Working site used to install, enable and exercise the migrated modules

A.2 Branching Convention

All work was carried out on a dedicated branch, `sli-intern`, created from the main development branch of each repository. The branch was pushed at the completion of each coherent unit of work rather than at the end of the internship, so that review feedback arrived while it could still be acted on cheaply.

```
1 git checkout -b sli-intern
2 git add <files for one logical change>
3 git commit -m "<what changed and why>"
```

```
4 git push origin sli-intern
```

Listing A.1: Branch creation and push cycle

A.3 Commit History

The following figures show the commit history of the `sli-intern` branch in each repository. Read together they trace the order of work described in Chapter 3: the FreeCAD Lab Migration module first, then DWSIM Custom Model, then DWSIM Flowsheet, and finally DWSIM Textbook Companion.

FreeCAD Lab Migration

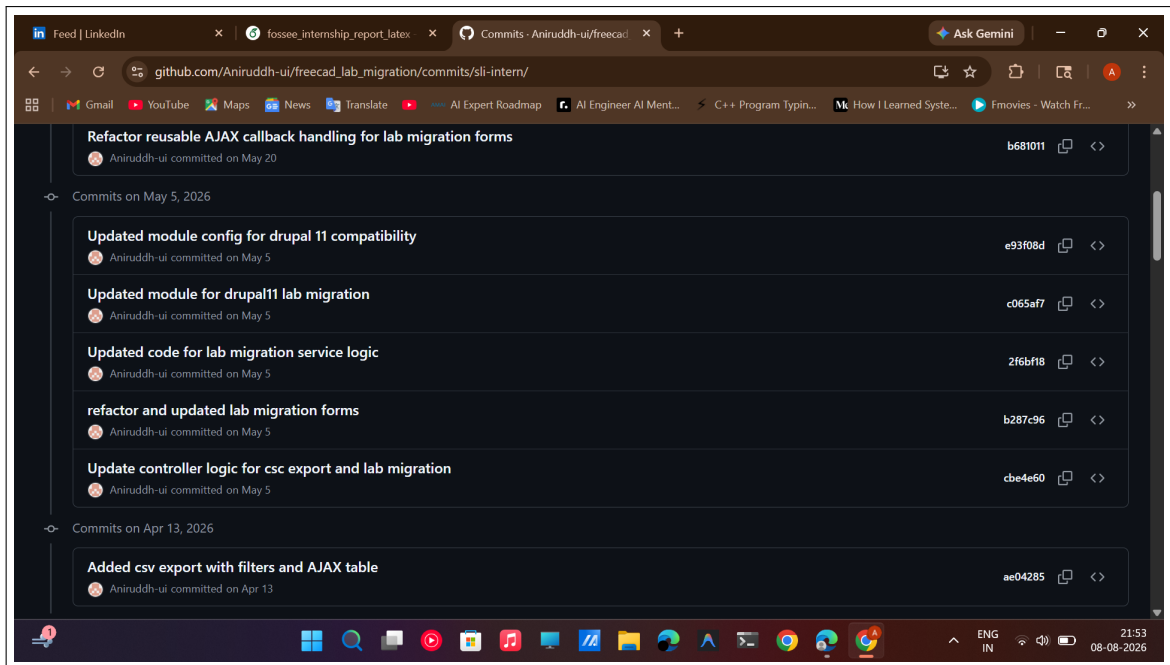


Figure A.1: `freecad_lab_migration`, branch `sli-intern`: Export CSV with filters and the AJAX table, followed by the module configuration, service logic and form migration work

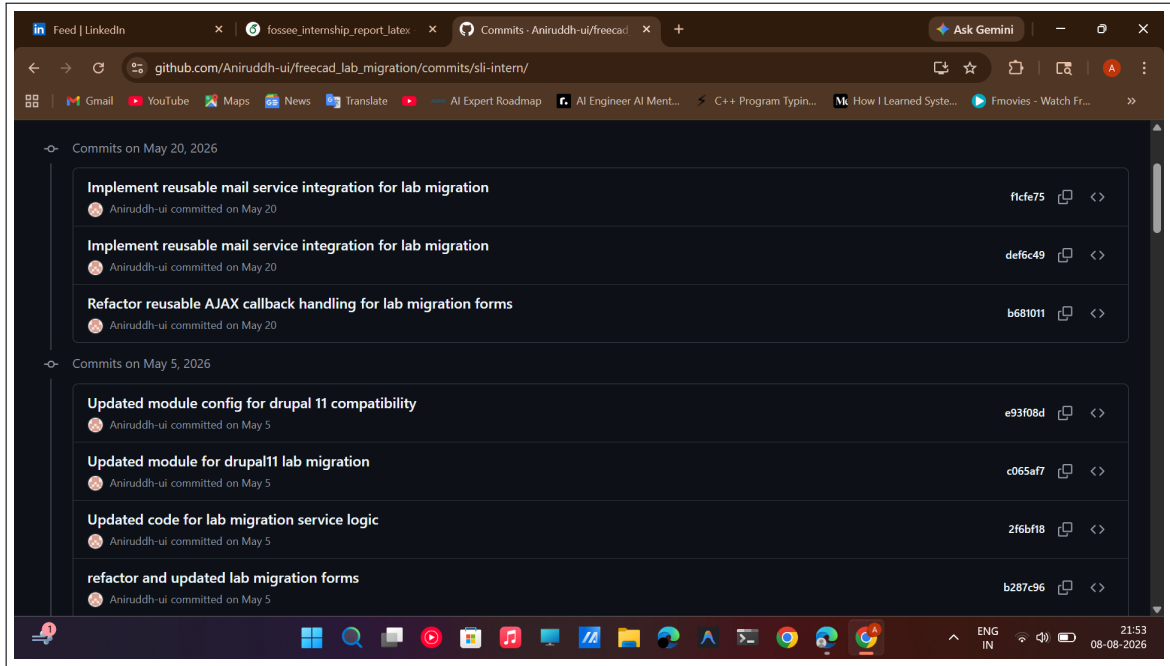


Figure A.2: freecad_lab_migration, branch sli-intern: reusable mail service integration and the refactoring of AJAX callback handling across the lab migration forms

DWSIM Custom Model

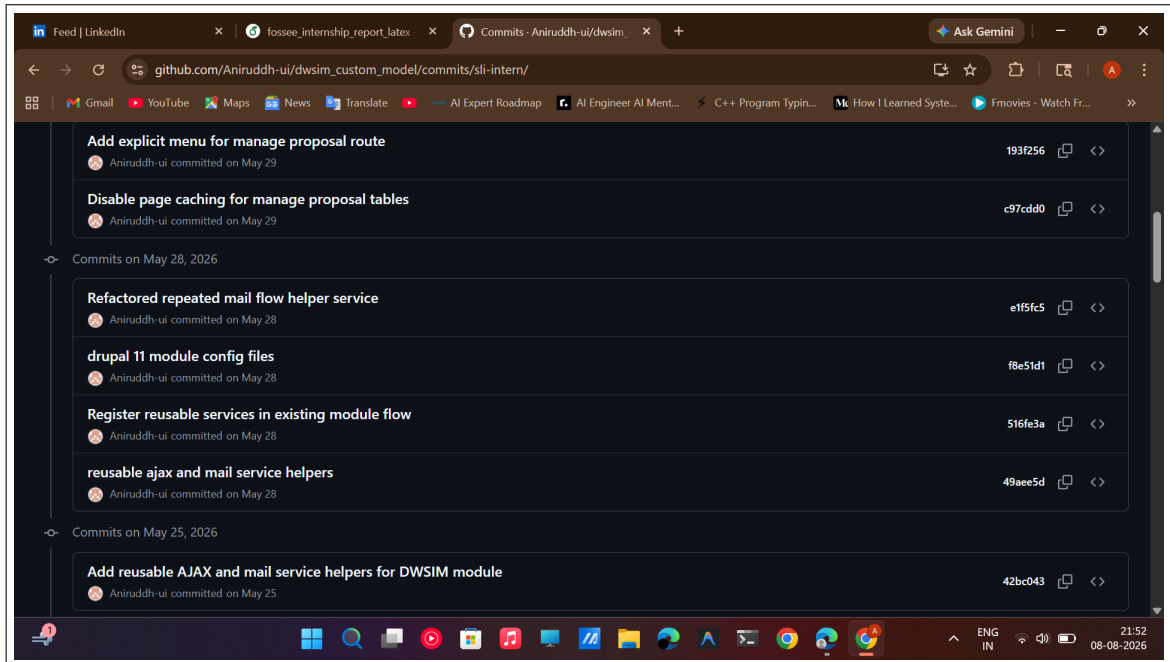


Figure A.3: dwsim_custom_model, branch sli-intern: the reusable AJAX and mail helpers introduced for the DWSIM module and registered into the existing module flow, alongside the Drupal 11 module configuration files

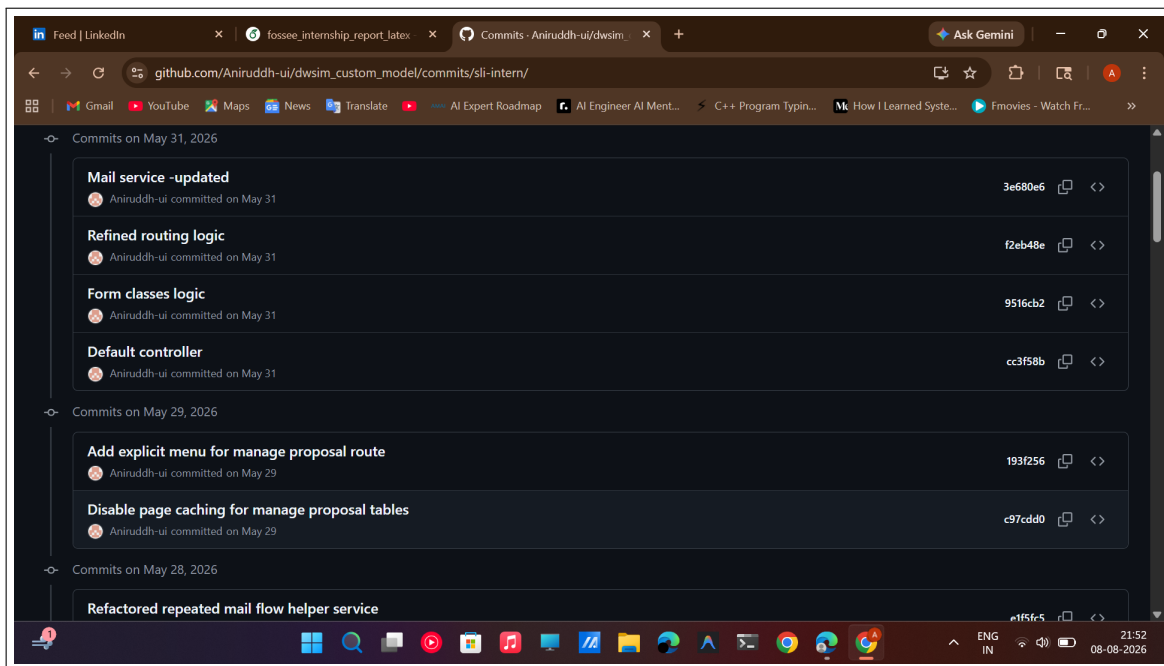


Figure A.4: `dwsim_custom_model`, branch `sli-intern`: controller, form classes, routing and mail service work, with the caching and menu adjustments made for the manage-proposal screens

DWSIM Site: Flowsheet and Textbook Companion

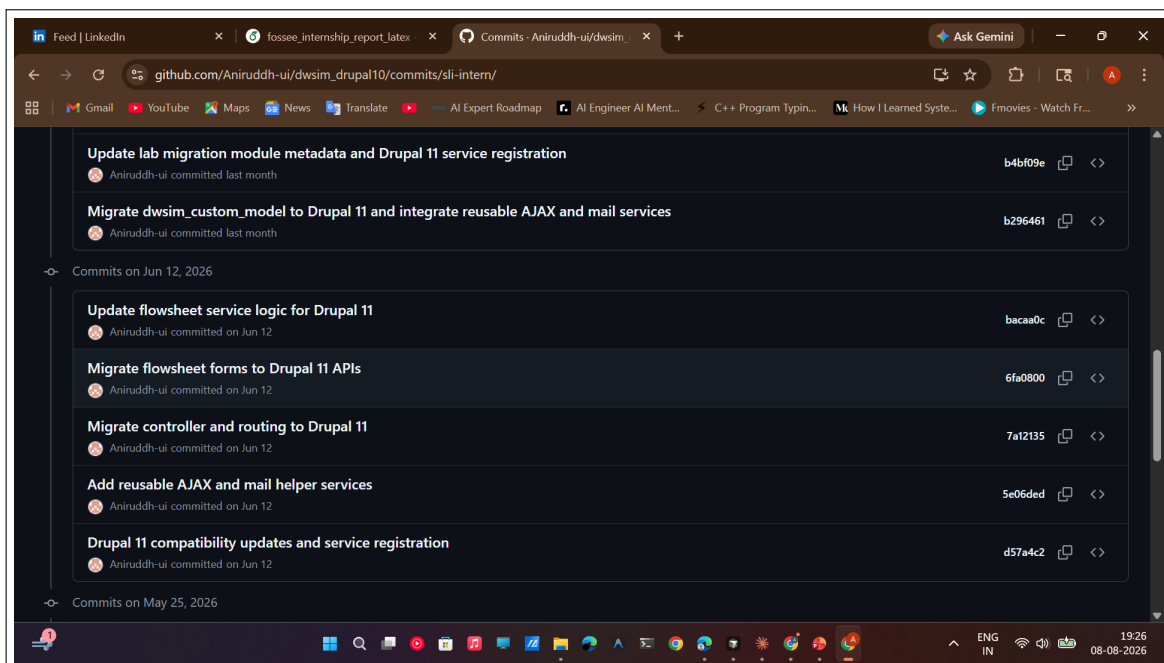


Figure A.5: `dwsim_drupal10`, branch `sli-intern`: DWSIM Flowsheet migration — forms, controller and routing moved to Drupal 11 APIs, with the reusable AJAX and mail helpers added and the services registered

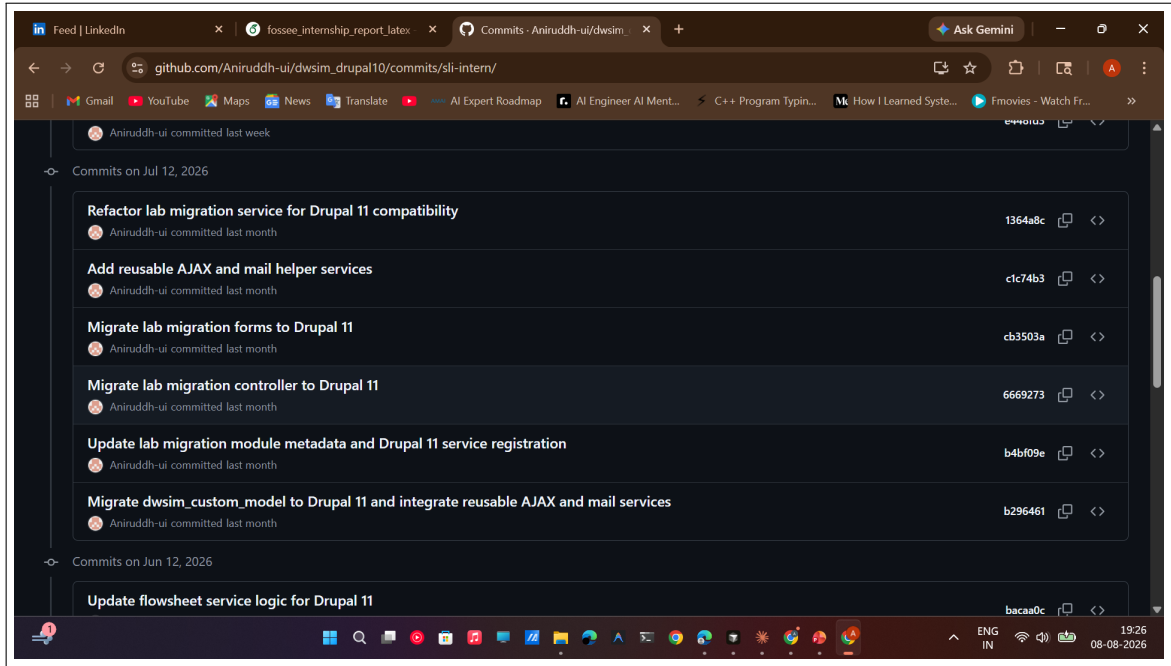


Figure A.6: `dwsimDrupal10`, branch `sli-intern`: lab migration module metadata and service registration, controller and form migration, and the integration of the reusable services into `dwsim_custom_model`

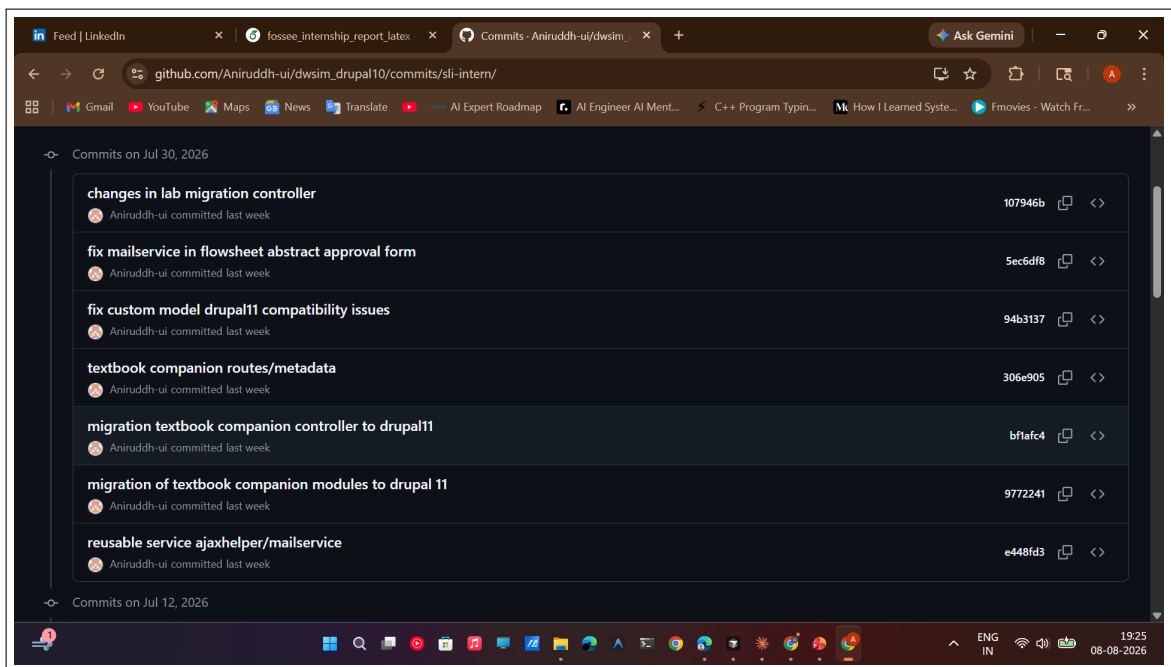


Figure A.7: `dwsimDrupal10`, branch `sli-intern`: the final phase — Textbook Companion modules, controller and routing migrated to Drupal 11, custom model compatibility fixes, and the flowsheet abstract approval mail service fix

Appendix B

Internship Timeline

The internship ran from February 2026 to July 2026. The table below summarizes the work by phase; the commit history reproduced in Appendix A is the corresponding record on the `sli-intern` branches. A day-by-day log was maintained separately in the shared progress spreadsheet.

Table B.1: Phase-wise summary of the internship

Period	Phase	Work carried out
February 2026	Onboarding and environment	Familiarisation with the FOSSEE Drupal applications; containerised development stack set up with Docker and Podman; Drupal 10 site installed locally from a database dump; Composer, Drush and Git workflow established
March 2026	Audit and first migration	Deprecated and removed API audit using Upgrade Status, Rector dry runs and direct source search; inventory compiled; FreeCAD Lab Migration module metadata widened to admit Drupal 11; mail, path argument and messenger call sites migrated
April 2026	Export CSV	Export CSV feature designed and implemented for the Lab Migration module with status and year filters, streamed response and an AJAX-driven preview table; controller logic updated to support it
May 2026	Reusable architecture and Custom Model	Lab migration module configuration, service logic and forms updated for Drupal 11; reusable mail service integrated and AJAX callback handling refactored across the lab migration forms; DWSIM Custom Model migrated, with the reusable AJAX and mail helpers registered into its existing module flow, and its controller, form classes and routing reworked
June 2026	DWSIM Flowsheet	Flowsheet forms, controller and routing migrated to Drupal 11 APIs; reusable AJAX and mail helper services added and registered; service logic updated; AJAX and <code>#states</code> interaction defects resolved
July 2026	Textbook Companion and completion	Lab migration service refactored for Drupal 11 compatibility and the reusable services consolidated; Textbook Companion modules, controller, routing and metadata migrated; custom model compatibility fixes and the flowsheet abstract approval mail service fix; full workflow and regression testing across all four modules; documentation and final report