# FLVS Coding Best Practices

# Contents

# Coding Guidelines

This document contains the coding guidelines for FLVS Educator Developers.

## Style Guide

### Variable and Function names

- Use meaningful and pronounceable variable names.
- Use the same vocabulary for variable and function names

```
function getCourseRoster(): CourseRoster
```

- Use searchable names

```
const oneDayInMilliseconds = 24 * 60 * 60 * 1000;

setTimeout(restart, oneDayInMilliseconds);
```

- Use explanatory variables

```
for (const [key, Value] of some_object) { ... // is
not clear

for (const [courseId, instructor] of courses) { ...
// is more clear
```

- Use camelCase for variable and function names. Define type clearly when possible.

```
let goodPet: string = "dog";
```

- Be as explicit as possible

```
const em = getEmail() // is confusing later

const emailAddress = getEmail() // is less so
```

- Don't add unneeded context

```
type Pet = {
 petType: string
 petColor: string
}

// Should be

type Pet = {
 type: string
 color: string
}
```

## Class Names

- Use PascalCase for class names

```
class Animal { .... }
```

Enums should be in their own files and the barrel file (index.ts) updated to export up the chain.

## Interfaces

- Use PascalCase for name
- Use camelCase for members

Interfaces should be in their own files and the barrel file (index.ts) updated to export up the chain.

## Types

- Use PascalCase for name
- Use camelCase for members

Types should be in their own files and the barrel file (index.ts) updated to export up the chain.

## NameSpaces

- Use PascalCase for name

## Enums

- Use PascalCase for name
- Use camelCase for members

Enums should be in their own files and the barrel file (index.ts) updated to export up the chain.

## Type-Only Imports and Exports

When importing or exporting types/interfaces, use the type statement. This ensures we only import or export the type annotations and declarations. For additional information refer to the TypeScript Documentation .

```
import type { SomeThing } from "./some-module.js";


export type { SomeThing };
```

## Vertical Space

Vertical whitespace in code should be treated like paragraphs in the written language. Where each line of code is a sentence and each block of code is a paragraph. Whitespace is free, be sure to make liberal use of it, but also don't overuse it.

What to separate?

- Shebang from Imports
- Imports from Code
- Functions
- Setup, Running, & Returning
- Getting a variable from is validation.

What to combine?

- Call sequences

**Example**

```
//Bad Example

#!/usr/bin/node

import assessmentGradebookRouter from
'./routes/app/assessmentGradebook.route';

import logoffRouter from './routes/app/logoff.route';

import enrollmentNotesRouter from
'./routes/app/enrollment-notes.route';

process.umask(0);

const compression = require('compression');

const MemcachedStore = require('connect-
memcached')(session);

const app = express();

const port = process.env.SYSTEMS_API_PORT || 3000;

// compress all responses

app.use(compression());

app.use(cors({

  allowedHeaders: ['Authorization', 'Access-Control-
Allow-Origin'],

  credentials: true,

  origin: JSON.parse(process.env.CORS_ALLOW_ORIGIN),

}));


//Good Example

#!/usr/bin/node
```

```
import assessmentGradebookRouter from
'./routes/app/assessmentGradebook.route';

import logoffRouter from './routes/app/logoff.route';

import enrollmentNotesRouter from
'./routes/app/enrollment-notes.route';


process.umask(0);


const compression = require('compression');

const MemcachedStore = require('connect-
memcached')(session);

const app = express();

const port = process.env.SYSTEMS_API_PORT || 3000;


// compress all responses

app.use(compression());


app.use(cors({

  allowedHeaders: ['Authorization', 'Access-Control-
Allow-Origin'],

  credentials: true,

  origin: JSON.parse(process.env.CORS_ALLOW_ORIGIN),

}));
```

## Iterating on an Array

When iterating over an array you should use a for loop instead of forEach as the callback function incurs a performance penalty. The readability of them are near identical.

```
    //Bad Example
    const array = [1,2,3,4,];
    array.forEach((item, index) => {
    ... Do some work with the array items.
    })

    //Good Example
    const array = [1,2,3,4,];
    for (const [index, line] of array.entries()) {
    ... Do some work with the array items.
    }
```

## Trailing Commas

When technically possible use trailing commas in your TypeScript / JavaScript code. This leads to easier updates along with cleaner commits. Note that JSON does not support trailing commas and adding them will produce invalid JSON.

```
//Bad Example
import {
  EnrollmentsService,
  MdbModifyEnrollmentInterface,
  PreflightEvaluationInterface,
  BuildPreflightDBCommandsInterface,
  PreflightDBCommands,
  ENROLLMENT_ERROR_CODES,
```

```
  PREFLIGHT_MSGS,
  Roles,
  MdbSelector,
  ComplexResults,
  SyllabusService,
  SyllabusType,
  Syllabus
} from "@educator-ng/common";

//Good Example
import {
  EnrollmentsService,
  MdbModifyEnrollmentInterface,
  PreflightEvaluationInterface,
  BuildPreflightDBCommandsInterface,
  PreflightDBCommands,
  ENROLLMENT_ERROR_CODES,
  PREFLIGHT_MSGS,
  Roles,
  MdbSelector,
  ComplexResults,
  SyllabusService,
  SyllabusType,
  Syllabus,
} from "@educator-ng/common";
```

*Bad JSON Example*

```
{
        "configuration_id":"use_legacy_urls",
        "_comment":"This config will determine which
features should use legacy URLs. The extra_data key
is the name of the navbar item and the value is the
legacy URL that will replace the configured URL. Any
routerLink properties for those navbar items will be
deleted.",
        "active":true,
        "installations":[
                {
                        "netapp":"flvs840",
                },
                {
                        "netapp":"f840",
                },
        ]
}
```

*Good JSON Example*

```
{
        "configuration_id":"use_legacy_urls",
        "_comment":"This config will determine which
features should use legacy URLs. The extra_data key
is the name of the navbar item and the value is the
legacy URL that will replace the configured URL. Any
routerLink properties for those navbar items will be
deleted.",
        "active":true,
        "installations":[
                {
                        "netapp":"flvs840"
                },
                {
                        "netapp":"f840"
                }
        ]
}
```

# General Frontend Principals

## Component-Based Architecture

Strive to break down complex UIs into small, reusable, and maintainable components. Each component should have a single responsibility.

## Smart Components

These components (also called containers) are concerned with how things work. They manage state, fetch data, and connect to services.

## Dumb Components

These components (also called Presentational Components) are concerned with how things look. They receive data via inputs (props) and emit changes via outputs (events). They are highly reusable.

## State Management

- Use local state that is only relevant to a single component (e.g., form inputs, toggles, etc.).
- Use shared/global state that needs to be accessible by multiple, non-related components (e.g., user authentication, notifications, etc.). Use dedicated state management libraries (NgRx for Angular, Redux for React) for complex global state.

# Modules and Lazy Loading

## Angular Lazy Loading

In Angular, lazy loading is typically implemented at the router level using loadChildren.

```
// app-routing.module.ts

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from
'@angular/router';

const routes: Routes = [ { path: 'dashboard', // Use
loadChildren to point to the feature module
loadChildren: () =>
import('./dashboard/dashboard.module').then(m =>
m.DashboardModule), // This module will only be
loaded when the user // navigates to the '/dashboard'
```

```
route. }, { path: 'profile', loadChildren: () =>
import('./profile/profile.module').then(m =>
m.ProfileModule), }, // ... other routes ];

@NgModule({ imports: [RouterModule.forRoot(routes)],
exports: [RouterModule] }) export class
AppRoutingModule { }
```

### React Lazy Loading

Lazy loading in React is achieved by using React.lazy() for components and React.Suspense to provide a loading fallback while the component is being fetched.

```
// App.js

import React, { Suspense, lazy } from 'react';

import { BrowserRouter as Router, Routes, Route }
from 'react-router-dom';

// Lazily import components const Dashboard = lazy(()
=> import('./routes/Dashboard')); const Profile =
lazy(() => import('./routes/Profile'));

// A component to show while lazy components are
loading const LoadingSpinner = () => Loading...;

const App = () => ( {/* Suspense provides the
fallback UI while components load */} <Suspense
fallback={}> <Route path="/dashboard" element={} />
<Route path="/profile" element={} /> );

export default App;
```

## Angular Best Practices

### Change Detection

Use OnPush change detection strategy for components to optimize performance. This tells Angular to only check for changes when @Input() references change, an event is fired from the component, or it's manually triggered.

### Service and Dependency Injection

Delegate business logic, API calls, and state manipulation to services. Use Angular's built-in Dependency Injection (DI) to provide these services to components.

### Async Pipe

Prefer using the async pipe in templates to subscribe (and automatically unsubscribe) from Observables. This prevents memory leaks.

```
// component.ts

import { Component } from '@angular/core';

import { DataService } from './data.service';

@Component({ selector: 'app-data-display', template:
> <div *ngIf="data$ | async as data; else loading">
<h1>{{ data.name }}</h1> </div> <ng-template
#loading>Loading data...</ng-template> }) export
class DataDisplayComponent { // Expose the Observable
directly data$ = this.dataService.getData();

constructor(private dataService: DataService) {} }
```

## React Specific Best Practices

### Functional Components and Hooks

Prefer functional components with Hooks (e.g., useState, useEffect, useContext) over class components for state management and side effects.

### Keys for Lists

When rendering a list of items using .map(), always provide a stable, unique key prop to each element. This helps React identify which items have changed, are added, or are removed.

### Memoization

Use React.memo for components, useMemo for values, and useCallback for functions to prevent unnecessary re-renders. Use them strategically, as overuse can add its own overhead.

### Component File Structure

Group related component files together. A common pattern is to have a folder for each component containing the component file (MyComponent.jsx), its styles (MyComponent.module.css), and its tests (MyComponent.test.js).

## Props

Keep props read-only. A component should never modify its own props. If a value needs to be changed, it should be managed as state and passed down.

# MongoDB Conventions

## Field Names

Field names in MongoDB should be lowercase with words separated by underscores. Any foreign key reference fields should end in "_idfk". Examples of both of these can be seen below.

```
{
    "_id" : ObjectId("6320c0a84caa3872c9b2182d"),
    "student_idfk" : ObjectId("613f98a15a0210147d3fe1f7"),
    "attachments" : null,
    "assessment_info" : {
        "flat_file_index" : NumberInt(14),
        "name" : "4.3 Quiz",
        "type" : "assignment",
        "label" : "Quiz",
        "segment" : NumberInt(1)
    },
    "dir" : "learn-qa",
    "submission_count" : NumberInt(1),
    "netapp" : "e1",
    "current" : {
        "instructor_idfk" : ObjectId("6137e2836a27355173e81cec"),
        "raw_points_possible" : NumberDecimal("25"),
        "gradebuilder_points_possible" : NumberDecimal("25"),
        "gradebuilder_points" : NumberDecimal("0"),
        "extracted" : ISODate("2022-09-13T12:40:56.000-0500"),
        "raw_points_earned" : NumberInt(0),
        "course_idfk" : ObjectId("613f98e65a0210147d3fe1fa"),
        "submission_status" : "Submitted"
    },
    "cid" : "4951",
    "enrollment_idfk" : ObjectId("613f98eb5a0210147d3fe1fc")
}
```

## Adding Interfaces to Collections

You should always add the interface to a collection when querying MongoDB or to the find Query

```
//Bad Example

const someCollection =
MongoDB.get2(process.env.GENERAL_MDB_DATABASE).collec
tion("someCollection");


//Good Example
```

```
const someCollection =
MongoDB.get2(process.env.GENERAL_MDB_DATABASE).collec
tion<SomeInterface>("someCollection");
```

Query Example

```
//Bad
  async getAssessmentSubmission(submissionId:
string): Promise<AsAssessmentSubmission> {
    return MongoDB.get()
      .db(process.env.ASSESSMENTS_MDB_DATABASE)
      .collection("as_assessment_submissions")
      .findOne({
        _id: MongoDB.objectId(submissionId),
      });
  }
//Good
  async getAssessmentSubmission(submissionId:
string): Promise<AsAssessmentSubmission> {
    return MongoDB.get()
      .db(process.env.ASSESSMENTS_MDB_DATABASE)
      .collection("as_assessment_submissions")
      .findOne<AsAssessmentSubmission>({
        _id: MongoDB.objectId(submissionId),
      });
  }
```

## Converting String ObjectIds to ObjectIds types

We have an objectId method in our MongoDB wrapper library, please use that. This is to help combat inconsistent use of calling the ObjectId method from the mongodb library, which is shown below.

```
//Bad Example

enrollmentObjectIds =
enrollmentIdfks.map(enrollmentIdfk => new
ObjectId(enrollmentIdfk));


//or

enrollmentObjectIds =
enrollmentIdfks.map(enrollmentIdfk =>
ObjectId(enrollmentIdfk));



//Good Example

enrollmentObjectIds =
enrollmentIdfks.map(enrollmentIdfk =>
MongoDB.objectId(enrollmentIdfk));
```

## One vs Many methods

When using Insert, Update, etc. please use the specific "One" or "Many" variants of the call. This is for a few reasons. First, it's clearer what the expectation is from reading if one record is being inserted/updated/etc. or if many are expected (or could). Second, the non-specific versions of the methods are removed in future MongoDB libraries, so this will help reduce our refactoring efforts later.

```
//Bad

  async getAssessmentSubmission(submissionId:
string): Promise<AsAssessmentSubmission> {

    return MongoDB.get()
```

```
      .db(process.env.ASSESSMENTS_MDB_DATABASE)
      .collection("as_assessment_submissions")
      .find({
        _id: MongoDB.objectId(submissionId),
      });
  }
//Good
  async getAssessmentSubmission(submissionId:
string): Promise<AsAssessmentSubmission> {
    return MongoDB.get()
      .db(process.env.ASSESSMENTS_MDB_DATABASE)
      .collection("as_assessment_submissions")
      .findOne<AsAssessmentSubmission>({
        _id: MongoDB.objectId(submissionId),
      });
  }
```