FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	A module for grading in LearnShell
Student:	Jaroslav Hampejs
Supervisor:	Ing. Jakub Žitný
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

LearnShell is a modular system for managing and performing exams with programming assignments in scripting languages, especially Shell. LearnShell currently offers basic functionality for grading which is only visible for teachers and students inside LearnShell.

Create a LearnShell integration with Grades service:
1. Analyze Grades API and pinpoint the methods that you'll use in your solution.
2. Analyze the current state and architecture of grading in LearnShell and propose the communication protocol for connection to Grades.
3. Implement the communication between LearnShell and Grades.
4. Make sure you document your code and cover it with tests (mainly unit tests).

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 10 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

A module for grading in LearnShell

Jaroslav Hampejs

Department of Software Engineering
Supervisor: Ing. Jakub Žitný

May 13, 2021

Acknowledgements

I would like to thank Ing. Jakub Žitný, the supervisor of this thesis, for his guidance and support. I would also like to thank my family and friends for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Hampejs, Jaroslav. *A module for grading in LearnShell.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Práce se zabývá vývojem nástroje pro komunikaci mezi klasifikačním systémem Grades a aplikací LearnShell. LearnShell, jakožto nezávislá vzdělávací platforma, spravuje svá data interně. Důsledkem je nedostupnost klasifikačních dat studentů v Grades. Navíc, tento problém se netýká pouze aplikace LearnShell.

Je předložen návrh a implementace modulu umožňující jednoduchou integraci Grades do jakékoli Node.js aplikace. Předložený návrh modulu je založen na rešerši a analýze Grades API a relevantních webových technologií. Implementovaný modul je dále podroben unit testování a integrován do testovací aplikace. Vyvinutý modul je veřejně a zdarma dostupný skrze NPM. Je přiložen manuál popisující instalaci, nastavení a použití modulu.

Klíčová slova klasifikace, Grades, LearnShell, SDK, API, NPM, modul, TypeScript, JavaScript

Abstract

This thesis focuses on developing a tool for communication between the Grades assessment system and the LearnShell application. LearnShell, being an independent educational platform, manages all its data internally. As a result, none of the LearnShell classification data is accessible via Grades. Moreover, this problem is not unique to LearnShell.

Design and implementation of a module allowing easy integration of Grades into any Node.js-based application is presented. The proposed design of the module is based on the research and analysis of Grades API and relevant web technologies. The implemented module is further subjected to unit testing; it is also integrated into a test application. The developed module is publicly available and freely obtainable via NPM. A manual describing the process of installation, setting-up, and usage is included.

Keywords assessment, Grades, LearnShell, SDK, API, NPM, module, TypeScript, JavaScript

Contents

List of Figures

List of Tables

Introduction

Managing a university course is not an easy task. Dozens or sometimes even hundreds of students can be associated with a single course at any given semester. Considering the number of assigned tasks throughout a course run, it is not difficult to imagine the enormous amount of data that needs to be managed. This data also includes student assessments.

Many web services and applications have been created to help with course organization, for example, KOS, Projects FIT, and Grades. In some cases, an application was created directly to support a specific course or a set of courses, for example, Progtest, MARAST, Grades, and LearnShell. Progtest supports courses taught by the Department of Theoretical Computer Science, whereas MARAST supports math-oriented courses taught by the Department of Applied Mathematics. Grades application provides a central hub for student assessments.

This thesis focuses on LearnShell – a web application developed to support the BI-PS1 (Programming in Shell) course. An inconvenience arose, however. Even though Grades was meant to be the center for student assessments, some applications do not synchronize the student data with Grades. Consequently, there is no single place for students to see their assessments regarding all their courses.

The goal of this thesis is to provide a way to connect independent educational platforms with Grades. The initial research focuses on the way student grades are currently managed at FIT CTU. Possible ways of remote communication and data storage are discussed with respect to the existing structure of LearnShell and Grades.

The analysis focuses on the Grades assessment system and its public API. Based on the analysis, the design of a tool for easy integration of Grades into LearnShell and other course-supporting applications is proposed. Finally, the implemented tool is tested using unit tests, and the distribution of the tool is discussed.

To summarize, the main goals of this thesis is to create a tool that:

- allows extending LearnShell and other educational platforms by providing Grades synchronization support,
- provides automatic authentication for communication with Grades, and
- offers a suitable interface for data exchange between Grades and LearnShell.

Chapter 1

Research

This chapter focuses on the means of electronic educational assessment at FIT CTU, including the Grades and LearnShell applications; a description of their structure, features, and usage is presented. Furthermore, this chapter touches on modular programming, RESTful web services, and popular web development frameworks.

1.1 Electronic Student Assessments at FIT CTU

As mentioned before, many web applications and learning environments have come into existence to enhance the learning experience at FIT CTU. Some of these applications work with assessment data. Grades application was built to help courses manage the student grades by creating an online center for all the assessment data at FIT CTU.

Unfortunately, most of the educational applications mentioned in the introduction (MARAST, Progtest, LearnShell) manage the assessment data independently. The assessments are either transferred to Grades manually or are not synchronized with Grades at all. More importantly, Grades provides a public API that can be used for synchronization with other applications.

1.1.1 Grades

This section describes the Grades assessment system, its structure, the data it manages, and the API it provides. The primary source of information used in this section is the API's documentation website [1].

Structure and Data Grades stores and manages a vast array of data, ranging from course metadata, definitions, and student groups, to student assessment data. It also manages user settings, documents, roles, expressions, notifications, course skills, statistics, user login, and external application access. Course management requires access only to the data listed below.

Course The course metadata includes the course name and code, the types of classes, completion conditions, and the number of credits students will receive after completing the course.

Definition hierarchy The course definition hierarchy contains data regarding all evaluated tasks in a given course. A single definition represents a single evaluable task. The most common tasks are homework, quizzes, tests, and final exams. Each definition in the hierarchy can hold the following data: definition identifier and name, task type, evaluation type, minimum value, minimum required value, maximum value, semester code, course code, and many more. Nested definitions are also supported.

Student groups Students of a course are divided into groups for easier management. An important group called *test students* is available for testing and development purposes.

Assessments The assessment data for a given task includes the name of the associated definition, value, note, timestamp, and the teacher's username.

The Grades data is stored in a remote database. The application-database communication is based on fundamental procedures called *CRUD* operations [2]. These operations are further described in section 1.3.

API Grades offers a public API. That means Grades' functionality can be integrated into other applications. The API is divided into fifteen controllers, each focusing on a specific part of the data Grades manages. Each controller provides several endpoints, each accessible via its own URL. In terms of managing the student assessments, the following three controllers described in the list below are essential.

Definition controller The definition controller manages course definitions. It can be used for managing individual definitions or the definition hierarchy at once. It can also be utilized for retrieving the following data:

- a single specific definition
- the definition hierarchy
- a list of definitions in a specified order

Student-Group controller The Student-Group controller manages the student groups of a given course. It can only be used for retrieving available data; no create, update, or delete functionality is provided.

Student-Classification controller The Student-Classification controller manages student assessments. It supports creating, updating, and retrieving the assessments in several forms:

- all assessments of all students in a given student group
- assessments of a specific task for all of the students in a given group
- all assessments of a specific student
- student's assessment overview

Not all resources are accessible publicly. For example, to gain access to the definition hierarchy, student groups, and classifications, it is first necessary to authenticate and obtain an access token that can then be used to access the desired resources.

1.1.2 LearnShell

LearnShell is an educational web platform focused on the BI-PS1 (Programming in Shell) course. This section presents the functionality of LearnShell and the way it currently handles assessments.

Capabilities LearnShell allows teachers to assign and evaluate tasks such as homework and tests. Students use the app to submit assigned tasks and to practice during class or at home. Additionally, LearnShell offers ways to automate the evaluation process for specific tasks.

Internal Assessments The assessment data is stored in the LearnShell database, accessible to the students only via the LearnShell web interface. The internal structure of the assessment data stored in LearnShell is presented in figure 1.1. The evaluable tasks in the LearnShell system include two kinds of entities – assignments and exams. A *Correction* entity representing the task evaluation belongs to each evaluable task. Therefore, it is important to consider Grades synchronization functionality for the assignments and exam assessment data.

1.2 Modular Programming

Modular programming is a software design technique based on the idea of connecting multiple software modules[1] into a more complex application. Separating the application code into modules can significantly speed up the development and enhance the sustainability of the application since each module can be easily updated, split, extended, or completely replaced without changing the rest of the application. Additionally, dividing the application into modules makes for easier debugging, allows for more effective team cooperation, enhances code reusability, and improves readability, reliability, and manageability of the code [3].

[1] pieces of software, each responsible for a particular functionality but not functional by itself

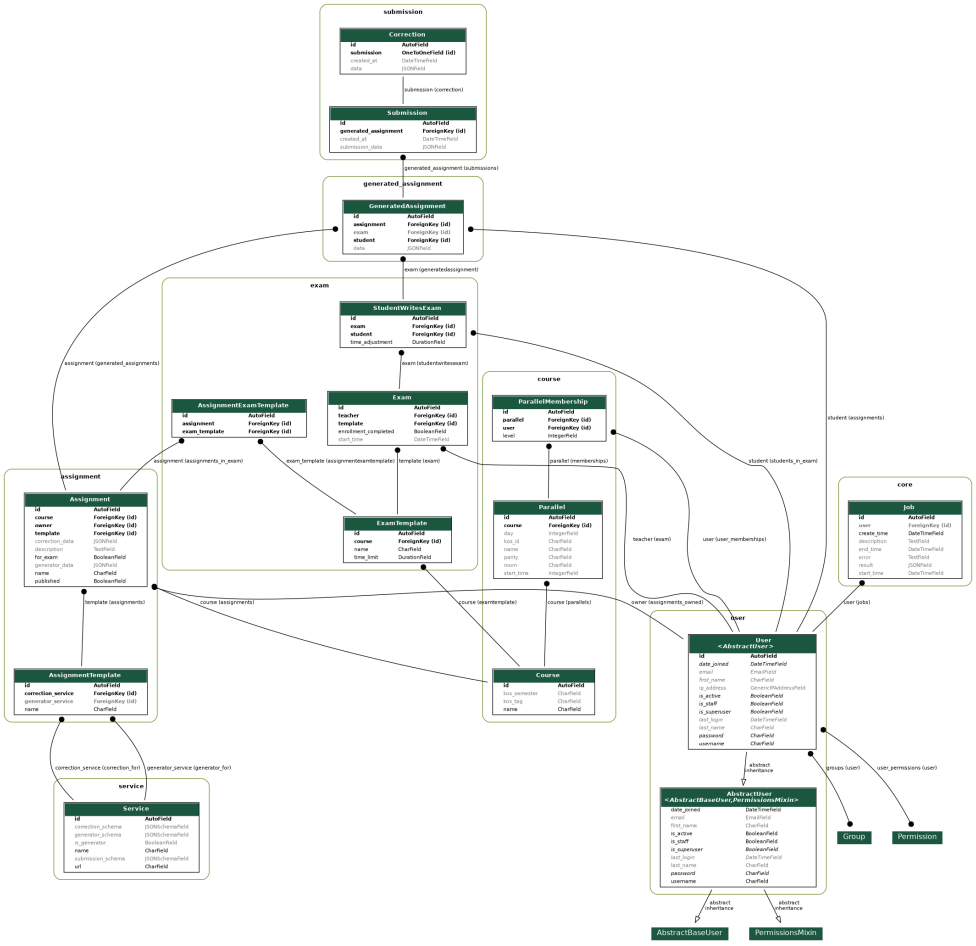


Figure 1.1: LearnShell database schema; sourced from the LS Core project at `https://gitlab.fit.cvut.cz/learnshell-2.0/ls`

1.2.1 Module design

To create a module, deciding on the following three points is essential: the technical details of the development (programming language, framework), the purpose of the module, and the way of distribution. The following paragraphs go into more detail about why these steps are necessary.

Programming language Deciding on the programming language to be used to develop the module is vital. Usually, the language chosen for the module corresponds with the language used to write the application the module is to be integrated into. In some cases, the languages may differ, however. An excellent example of this is a computer graphics software called Blender, a complex application consisting of many different parts and modules written in C, C++, and Python [4].

Main function To ensure a smooth development process, it is critical to define the module's purpose. The functionality of the developed module should be pretty easy to explain; otherwise, it is necessary to divide the module into several sub-modules.

Distribution The distribution method depends on the module programming language and environment.

Finding the correct answers to these three key questions should simplify the subsequent development and ensure the module's quality and ease of use.

1.3 RESTful web services

There are a few common approaches for dealing with remote interfaces. A typical architecture used for remote resource management is REST, based on a set of operations called CRUD. This section presents the fundamentals of REST and CRUD, the connection between the two, and their relevance to this thesis.

1.3.1 CRUD

CRUD is a set of operations used for working with remote data sources [5]. The set consists of four operations:

CREATE creates a new resource in the persistent storage

READ (sometimes **RETRIEVE**) retrieves a desired resource from the persistent storage

UPDATE updates the resource in the persistent storage

DELETE removes the resource from the persistent storage

1.3.2 REST

REST, or Representational State Transfer, is an architecture that allows a simple realization of the CRUD operations described earlier [6]. HTTP requests and responses are used to manage remote resources. One of the main parameters of the HTTP request is a *method*. The *method* specifies the desired action to be performed with the given resource. Various *methods* can be specified: OPTIONS, HEAD, GET, POST, PUT, DELETE, TRACE, CONNECT [7]. Not all the mentioned *methods* are relevant to this thesis. A description of the relevant methods and their mapping to the CRUD operations follows:

POST for the CREATE operation

GET for the READ/RETRIEVE operation

PUT for the UPDATE operation

DELETE for the DELETE operation

When an application wants to work with a remote resource, it sends an HTTP request to the server that manages it. The server processes the request and sends a response back to the application. Using HTTP as the mean of communication makes the whole system very standard and straightforward, thanks to the request/response way of communication [6]. Following are examples of an HTTP request and a response.

Basic HTTP request example [8]:

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0
Host: localhost:8000
Accept: application/json
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

Basic HTTP response example [8]:

```
HTTP/1.1 200 OK
Server: Apache
Date: Wed, 10 Aug 2016 07:45:10 GMT
Content-Type: application/json
Connection: Keep-Alive
Content-Length: 190

{name: Jane, surname: Reynolds, ...}
```

REST vs RPC There are other ways of working with remote resources and web services, for example, XML-RPC (SOAP) or JSON-RPC. These alternatives are based on the RPC (Remote Procedure Call) technology used to execute routines on a remote computer – a client computer or a server [9]. Some shortcomings of RPC have led to more widespread REST support in typical use cases [10].

1.3.3 REST API

An application's API (Application Programming Interface) is a communication layer that exposes the application functionality to the outside. REST API (or RESTful API) is an interface that conforms to the REST architecture, which consists of the constraints listed below [11].

Client-Server architecture Decoupling the client and server sides of the communication creates a system where the functional responsibilities of the server and the client are separated. Moreover, many different clients can use the services provided by the server-side.

Statelessness To achieve *stateless* communication, every request passed from the client to the server must be independent and understandable without further context. A positive effect of this approach is server load reduction resulting in a performance increase.

Cacheability The requests have to be labeled either as cacheable or non-cacheable. With this information, some of the server-client interactions can be eliminated, which leads to a further performance increase.

Layered system Layering allows other components to be present between the client and the server, for example, proxy servers or load balancers. This approach is transparent to both the client and the server and allows for further scalability and security enhancements.

Uniform interface Keeping the interface uniform helps decouple the architecture, meaning each part or component can evolve independently.

1.4 Communication with REST APIs

Interacting with a REST API always involves raw HTTP requests; however, the low-level request/response handling is managed using various libraries in development. This section provides a closer look at communication with REST APIs from a JavaScript/TypeScript environment since these are amongst the most popular languages for web development. Three possible ways of managing and handling HTTP requests and the respective responses are described in the following sections.

1.4.1 XMLHttpRequest

The most basic communication with an API via raw HTTP is using XMLHttpRequests (or XHRs). XHR is a JavaScript object that handles the interactions between a browser and a server. The standard XHR JS object supports both synchronous and asynchronous requests, the latter with the use of callbacks, a JS technique of passing functions as parameters to other functions [12, 13].

1.4.2 Fetch API

The Fetch API is a native JS API based on the *Promise* technology used for making asynchronous requests and handling the responses. Leveraging

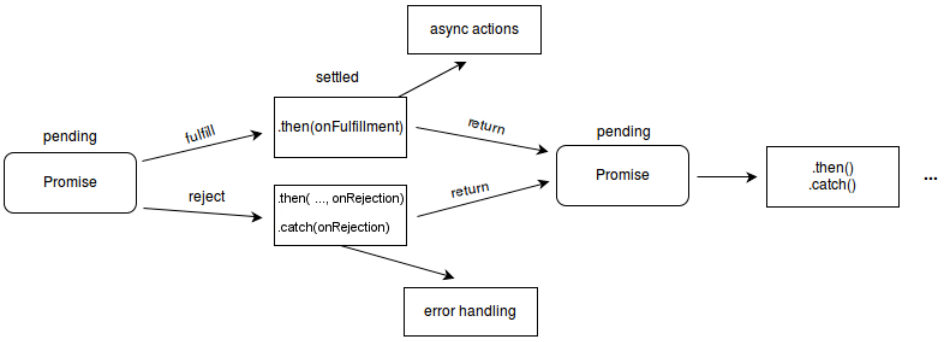


Figure 1.2: Promise states and chaining

Promises eliminates the necessity to deal with callbacks directly. More detailed information about the *Promise* technology is provided in section 1.4.3. The Fetch API is well supported across browsers and provides an easy-to-use object-based interface and a *fetch-then* syntax [14, 15].

1.4.3 Axios

Axios is a JavaScript library for asynchronous HTTP request management. It creates an additional layer between an application's code and the low-level XMLHttpRequests, providing the developer with a simple, easy-to-use interface for interacting with REST APIs. For this thesis, the following are the most crucial properties of Axios:

Promise API support Axios is a Promise-based library; it handles the success or failure of asynchronous operations via *Promise* objects. The *Promise* object is an abstraction over the state of an asynchronous response and its potential value. There are three possible states of a *Promise* object [16, 17]:

- pending (initial state after initialisation)
- fulfilled (request was successful)
- and rejected (request failed)

Promises can be chained, as shown in figure 1.2 [17].

Isomorphism Being isomorphic allows Axios to run both on the back-end and in the browser (front-end) using the same code base, resulting in broader support and flexibility.

Automatic JSON transformations Axios can automatically transform received text data into JSON objects and vice versa, potentially eliminating dozens of lines of identical code.

Axios supports two syntax forms: a *then-catch* syntax similar to the Fetch API and an *async/await* syntax usable within asynchronous functions. Using *async/await* can enhance readability, thanks to the elimination of many brackets.

Fundamental technologies of LearnShell and Grades This chapter introduces technologies and frameworks used within Grades and LearnShell applications, specifically Python, JavaScript, TypeScript, Django, Next.js, React, REST and CRUD.

1.5 Python

Python is an interpreted dynamically typed programming language supporting multiple programming paradigms, for example, object-oriented programming, procedural programming, and functional programming. In recent years, Python has become very popular in the AI (artificial intelligence) industry. It is also quite popular in the web development sector for creating the server side of applications and services [18].

1.5.1 Python Package Installer (pip)

Pip is a package installer for Python. It provides access to numerous packages for Python applications, helping to accelerate development by providing developers with ready-to-use tools. Pip itself allows looking up, installing, maintaining, and uninstalling the packages. It also manages the dependencies of the packages to ensure proper functionality.

1.6 Django

Django is an open-source framework for creating web applications. It is maintained by a non-profit organization called Django Software Foundation. The development of web apps using Django is driven by the MTV (Model-Template-View) architecture. MTV consists of three main layers.

Model The model layer contains Python classes that define the structure of given object types. It can generally be said that these classes represent rows from a database [19].

Template The template layer contains static templates for creating the underlying HTML for the final pages. Included with the framework is a templating language called DTL (Django templating language) that allows dynamic inclusion of data into the templates [20].

View The view layer contains classes and functions representing the business logic. This layer accepts requests from the client-side, ensures the request gets processed, and that a response is created and send back [21].

The back-end of the LearnShell project, called LS Core, was implemented using the Django framework. The source code of the project is available on GitLab[2].

1.7 JavaScript & TypeScript

JavaScript is a scripting language that can run on both the back-end (server-side) and the front-end (browser), making it a flexible web development technology. Being a scripting language, it does not require any compilation or transpilation. Moreover, numerous frameworks and libraries are available for JavaScript, providing developers with countless extensions for their projects [24].

TypeScript TypeScript is a so-called superset of JavaScript. It is an optionally typed language with full JavaScript support, providing strict control over the code where necessary. However, since it is a typed language, it needs to be transpiled into JavaScript by a TypeScript compiler [25].

1.7.1 Node Package Manager (NPM)

NPM is a package manager for JavaScript. Just like Pip, NPM provides web developers with an extensive library of modules. The NPM service consists of three main components [26]:

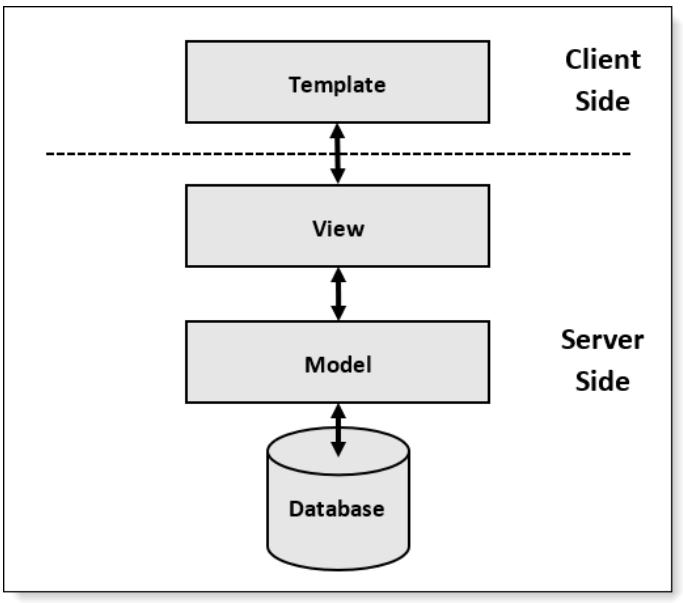
The website The website allows browsing the available packages, creating a user profile, and managing custom packages.

The CLI The CLI (Command Line Interface) is the primary way of interacting with the service for most users, used for publishing an updated version of the user's custom package or installing other packages.

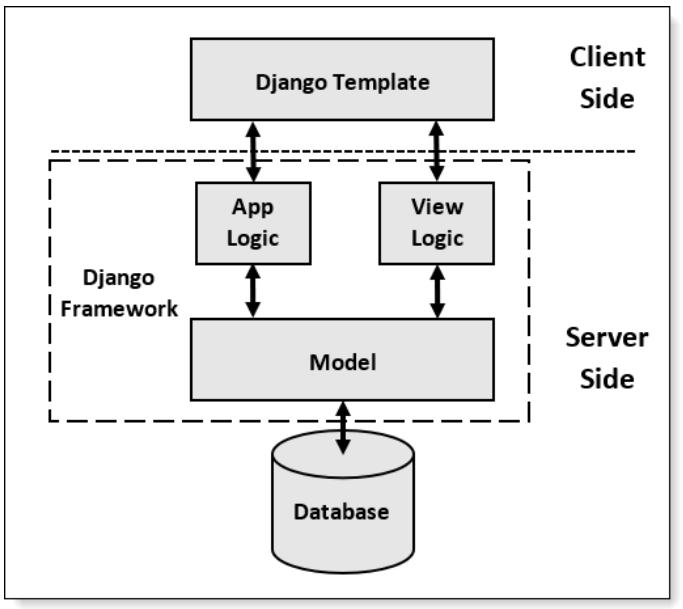
The registry The registry is a database of the available packages, both free and paid, and their metadata, such as links to the developer's website, documentation of the package, installation instructions, and statistics about its popularity.

All packages available through NPM are applicable in the Node.js environment and other environments based on Node.js. An example worth mentioning is the Next.js framework presented in section 1.8.

[2] see `https://gitlab.fit.cvut.cz/learnshell-2.0/ls`



(a) A simpler view at MTV [22]



(b) Detail of the MTV architecture [23]

Figure 1.3: Basic MTV architecture of a Django project

1.8 React & Next.js

React

React is a JavaScript library for creating the user interface of web applications. It can be used for single-page application development. Such applications do not require reloading the web page to change the displayed data. Instead, the framework utilizes the advantages of JavaScript and allows re-drawing only the elements that need to be updated. This approach can save both time and resources. Being built on JavaScript and easy-to-use flexible declarative code blocks called components has helped React reach the top of the list of the most favorite web frameworks [27].

Next.js

Next.js is a React-based framework for developing web applications. It wraps the React library discussed in the previous section, giving it an additional layer of abstraction that makes specific tasks more manageable, such as routing, SEO, broader CSS support, and remote data source communication. On top of that, Next.js brings server-side rendering [28, 29]. The framework is based on the Node.js platform, and it supports the use of modules available through NPM. The front-end of the LearnShell project (LS Web) was built using the Next.js framework and utilizes some of its advanced features like server-side rendering. The source code of the project is available through GitLab[3].

[3] see `https://gitlab.fit.cvut.cz/learnshell-2.0/ls-web`

Chapter 2

Analysis

After researching the means of electronic assessment at FIT CTU, modular programming technique, and the core technologies of Grades and LearnShell, it became apparent that the primary goal of this thesis would be to design, implement and test a module that could be integrated into web applications to provide Grades support.

The next step is to analyze the possible ways of developing and distributing the module. This includes selecting a suitable programming language, defining the module's functionality, and finding the right way of distribution. These aspects are discussed further in this chapter.

2.1 Selecting a Language

This section focuses on selecting a suitable programming language for developing the module. Python and JavaScript/TypeScript come to mind, as these are amongst the most popular languages used for web development. For the reasons noted below, TypeScript was chosen to be the development language for the module:

- ability to run on both the front-end and the back-end is easy to achieve using TS; using Python for this purpose would be unimaginable
- integration into the LearnShell platform (the LS Web project) requires compatibility with TS or JS
- type support

2.2 Defining the Functionality

The module will provide a communication channel with Grades. Based on the internal structure of LearnShell presented in section 1.1.2, the communication will mainly involve management of the definition hierarchy and student assessments.

2.3 Connecting Grades

The only way to interact with Grades from an external module is by using its public REST API, which involves working with HTTP requests, as mentioned in the research section 1.4. It is advisable to use an HTTP client, such as Axios. This library is easy to use, has a broad user base, and offers a modern interface based on *Promises*.

2.4 Means of Distribution

Since it was decided that the module will be developed in TypeScript, NPM is a straightforward choice for publishing and distribution. NPM provides tools for easy maintenance (pushing module updates) and guarantees broad availability.

2.5 Functional Requirements

The following are the functional and non-functional requirements for the module.

F1 – Provide access to course metadata The module will provide course metadata for a given course.

F2 – Provide access to course's definition hierarchy The module will provide access to the definition hierarchy of a given course. More specifically, the following operations regarding course definitions will be supported:

1. **getting a definition** The module will be able to retrieve a specified definition from the definition hierarchy.
2. **getting definition hierarchy** The module will enable retrieving the definition hierarchy of a specified course.
3. **getting all definitions** The module will be able to retrieve a list of all the definitions belonging to a specified course.

4. **adding a definition to definition hierarchy** The module will be able to add a definition into the definition hierarchy of a specified course.
5. **updating a definition** The module will allow updating individual definitions.
6. **and removing a definition** The module will allow deleting a specified definition.

F3 – Provide semester codes for current, previous and following semester The module will be able to provide current, previous, and following semester codes.

F4 – Provide a list of student groups of a course The module will be able to retrieve the student groups of a specified course.

F5 – Provide student assessment data The module will provide access to student assessment data. The following operations will be supported:

1. **getting all assessments of a student group** The module will be able to retrieve all assessments of the students belonging to a specified group.
2. **getting a specific assessment from every student of a group** The module will be able to retrieve a specific assessment for every student of a specified group.
3. **getting all assessments of a student** The module will allow retrieving a list of all assessments belonging to a specified student.
4. **adding student assessments** The module will enable adding new assessments to a specified student.
5. **updating student assessments** The module will allow updating individual assessments.

2.6 Non-functional Requirements

N1 – Availability in the form of an NPM module The module will be freely available through NPM for use in Node.js environments.

N2 – User Documentation A user manual will be provided with the module.

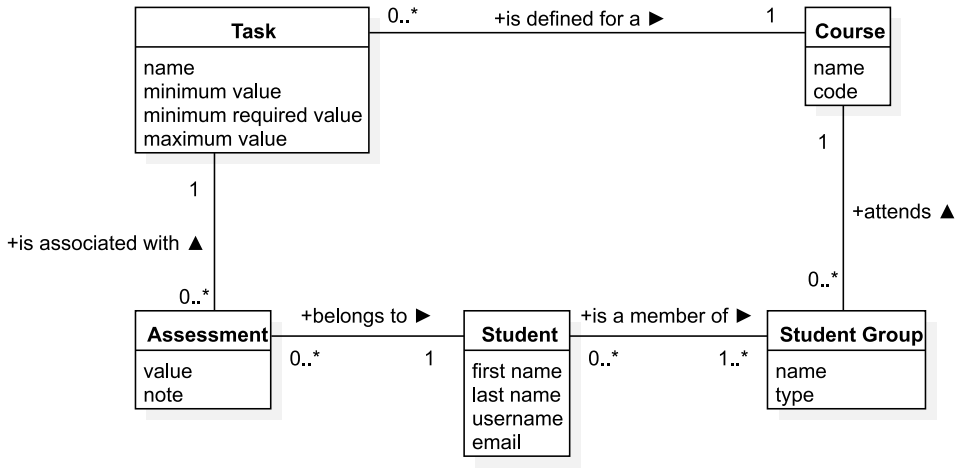


Figure 2.1: Simplified diagram of the relevant domains

2.7 Domain Model

Diagram 2.1 showcases the domain model of electronic assessment at FIT CTU. The scheme is relatively simple and consists of five domain classes. Each student is a member of one or more student groups in a particular course. A task represents an exam, homework, a quiz, or another evaluable assignment, and the student receives an assessment for each submitted task.

2.8 Summary and Goal

The analysis included defining the functionality of the module and listing the functional and non-functional requirements. A final form of the module was derived; the module will be written in TypeScript and distributed via NPM.

To summarize, the main goal of this thesis is to design, implement, and test a TypeScript NPM module that provides communication with Grades API and is compatible with Node.js environments.

Chapter 3

Design

The module's main functionality is mediating the communication between the Grades API and applications into which the module is integrated. A simple depiction of such structure can be seen in figure 3.1.

Grades API offers many endpoints to interact with, organized into controllers. To some extent, the endpoints reflect the most common actions necessary to manage a course, including the management of definition hierarchy, student groups, and student assessments. An overview of the API controllers

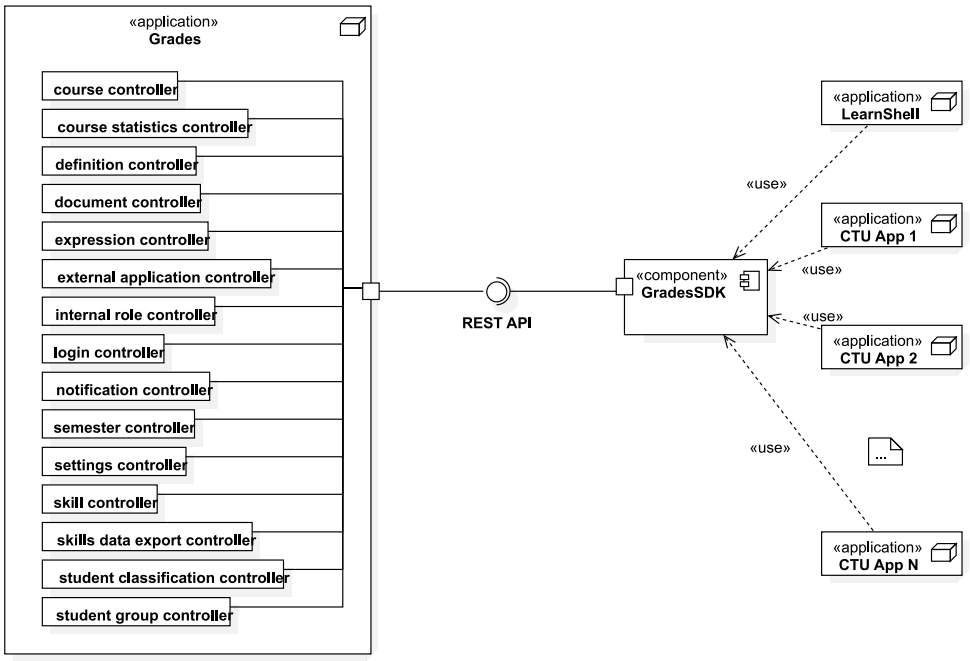


Figure 3.1: GradesSDK module integrated into applications

Grades API Controller	Relevant for Domain CLS	DEF	OTH	Description
course-controller	•	•	✓	Provides course information
course-statistics-controller	•	•	•	Provides statistics related to courses
definition-controller	•	✓	•	Manages course's definitions
document-controller	•	•	•	Provides documentation
expression-controller	•	•	•	Provides expression management
external-application-controller	•	•	•	Manages external applications
internal-role-controller	•	•	•	Manages internal roles
login-controller	•	•	•	Provides requesting user info
notification-controller	•	•	•	Manages notifications
semester-controller	✓	✓	•	Provides semester code validation
settings-controller	•	•	•	Manages settings
skill-controller	•	•	•	Manages skills
skills-data-export-controller	•	•	•	Skills Data Export Controller
student-classification-controller	✓	•	•	Manages classification of students
student-group-controller	✓	•	•	Manages student groups

Table 3.1: Grades API Controllers Relevance (CLS refers to classifications, DEF to definitions, OTH to other functionality)

is shown in table 3.1. The table also marks the controllers directly or indirectly related to managing the resources relevant to the subject of this thesis.

Based on table 3.1, these controllers will be needed for basic course management: course-controller, definition-controller, semester-controller, student-classification-controller and student-group-controller. Table 3.2 lists all the endpoints of the selected controllers. Based on the functional requirements presented in section 2.5, the endpoints leveraged by the module are highlighted.

In order to organize the endpoints highlighted in table 3.2, it is convenient to split them into three sets. First, a set of endpoints regarding the course definitions; second, a set of endpoints regarding the student assessments; and third, a group of endpoints regarding other necessary functionalities like managing course and semester codes.
The endpoints in each set require the course code parameter. Suppose each endpoint is mapped to a single function; it is necessary to supply the same parameters to each call. Instead of having many functions requiring the same parameters, classes will be utilized to store the shared data and make it available to all their methods representing the endpoint calls. The primary classes are described in the following list.

CourseDefinitions The `CourseDefinitions` class manages the definition hierarchy. Its public methods correspond to the selected Grades endpoints regarding the definitions.

CourseClassifications The `CourseClassifications` class contains public methods corresponding to the selected Grades endpoints regarding student assessments.

Grades API endpoints of all the relevant controllers	F1	F2	F3	F4	F5
GET .../{course-code}/information	✓	•	•	•	•
GET .../information	•	•	•	•	•
GET .../{course-code}/definition-hierarchy	•	✓	•	•	•
POST .../{course-code}/definition-hierarchy	•	✓	•	•	•
GET .../{course-code}/definitions	•	✓	•	•	•
PUT .../{course-code}/definitions	•	✓	•	•	•
GET .../{course-code}/definitions/{identifier}	•	✓	•	•	•
DELETE .../{course-code}/definitions/{identifier}	•	✓	•	•	•
PUT .../{course-code}/definitions/order	•	•	•	•	•
PUT .../{source-course-code}/definitions/clones/{target-course-code}/	•	•	•	•	•
GET /public/semester-code	•	•	✓	•	•
GET .../{course-code}/group/{group-code}/student-classifications	•	•	•	•	✓
GET .../{course-code}/group/{group-code}/student-classifications/{identifier}	•	•	•	•	✓
GET .../{course-code}/student-classifications	•	•	•	•	•
PUT .../{course-code}/student-classifications	•	•	•	•	✓
GET .../{course-code}/student-classifications/{student-username}	•	•	•	•	✓
GET .../{course-code}/student-classifications/{student-username}/latest	•	•	•	•	•
GET .../classification-overview/{student-username}	•	•	•	•	•
GET /public/course/{courseCode}/student-groups	•	•	•	✓	•

Table 3.2: Mapping of the Grades API endpoints to the functional requirements

Course The `Course` class stores the course code and provides instances of `CourseDefinitions` and `CourseClassifications` classes to manage the respective resources. It also provides the course metadata and student groups.

These and other building blocks of the module are further described in section 3.1 and depicted in figures 3.2, 3.3, and 3.4.

3.1 Module Structure

A total of nine classes and nine interfaces will make up the module. The following sections describe their structure and functionality. For more detail, see figures 3.2, 3.3, and 3.4.

3.1.1 Classes

GradesSDK The `GradesSDK` class is the main entry point to the module's functionality. It contains two public methods that are responsible for the basic operations. A more detailed description of the class can be seen in figure 3.2.

getCourse Provides a `Course` instance that can be used to manage course definitions and student assessments.

getSemesterCode Provides basic functionality for fetching the semester code, including getting the current, following, and previous semester codes and checking the validity of a provided semester code.

Course The `Course` class represents a single course registered within Grades, and it is the main access point for managing the course definitions and student assessments using the following methods. For more details, see figure 3.2.

code Returns the course code as a string value.

classifications Returns a `CourseClassifications` instance for managing the student assessments regarding the course.

definitions Returns a `CourseDefinitions` instance for managing the definition hierarchy of the course.

getCourseInfo Returns a `CourseInfo` object containing the course metadata.

getStudentGroups Returns an array of `StudentGroup` objects representing the student groups associated with the course.

CourseClassifications The `CourseClassifications` class manages student assessments for a given course. Its interface consists of the following methods. See figure 3.2 for more details.

getStudentClassifications Returns a `Student` instance containing information about the student and their assessments regarding the course.

getStudentClassificationsForGroup Returns an array of `Student` instances representing the member students of a given group with their assessments.

getStudentClassificationsForGroupAndDefinition Returns an array of `Student` instances, each containing a single `Classification` instance corresponding to the specified definition.

saveStudentClassifications Uses the provided `Classification` instances and transforms the contained data into objects that implement the `ClassificationDTO` interface before sending the processed data to Grades.

CourseDefinitions The `CourseDefinitions` class can be used to manage the course's definition hierarchy using these public methods. See figure 3.2 for more details.

createDefinitions Uses the provided `Definition` instance(s) and transforms the contained data into objects that implement the `DefinitionDTO` interface before sending the processed data to Grades.

getDefinition Returns a `Definition` instance representing the desired definition from the course definition hierarchy.

getDefinitionList Returns an array of `Definition` instances representing all definitions that make up the course definition hierarchy.

getDefinitionHierarchy Returns an array of `DefinitionWithChildren` instances representing the definition hierarchy of the course.

updateDefinitions Updates the course definition hierarchy based on the provided `Definition` instances(s).

deleteDefinitions Deletes definition(s) from the course definition hierarchy based on the provided identifier or `Definition` instances(s).

ApiRequestor The `ApiRequestor` class is used for the internal purposes of the module. At the base of almost every functionality provided by the module is an HTTP request. In order to avoid repetitive code and simplify the development of the module, the `ApiRequestor` class provides four methods regarding networking and communication with Grades API. These four methods are: `get`, `put`, `post`, and `delete`. Each of these provides a way of creating and sending an HTTP request of the respective method using the Axios package mentioned in section 1.4.3. Details about the structure of the `ApiRequestor` class can be seen in figure 3.2.

Definition The `Definition` class represents a definition from the course definition hierarchy. It implements the `DefinitionInterface` interface described in paragraph 3.1.2 and therefore contains all its properties. See figure 3.3 for more details.

DefinitionWithChildren The `DefinitionWithChildren` class represents a definition from the course's definition hierarchy, together with its child definitions in the `children` property. An `addChild` method is provided to enable adding child definition elements. See figure 3.3 for more details.

Student The `Student` class contains data about a student and their assessments regarding a given course. This class implements the `StudentInterface` interface described in paragraph 3.1.2 and therefore contains all its properties. More details available in figure 3.3.

Classification The `Classification` class represents a single assessment (in other words, an evaluated task) of a given student. The class implements the `ClassificationInterface` interface described in paragraph 3.1.2 and therefore contains all of its properties. See figure 3.3 for more details.

3.1.2 Interfaces

Several object interfaces are also part of the module to enhance usability and ensure the necessary data is present.

CourseInfo The `CourseInfo` interface describes a structure of an object representing the metadata of a given course. The interface consists of the following properties. See figure 3.2 for more detail.

courseName a string value; the name of the course in a language specified in the request

courseCode a string value; the code of the course in the standard format

classesType a string array; holds the class types of the course

completion a string value from a predefined list; describes the type of course completion

credits a number value; the number of credits the student will be awarded for successful completion of the course

DefinitionInterface The `DefinitionInterface` interface describes a structure of an object representing a single definition from the definition hierarchy of a given course using the following properties. More details can be found in figure 3.3.

parentIdentifier a string value; holds the identifier of a parent definition

semesterCode a string value; holds the semester code for which the definition is valid

namesForLocales an object containing the name/title of the definition optionally in multiple languages (Czech or English)

minimumRequiredValue a number value describing the lowest score required to pass the respective task

minimumValue a number value describing the lowest possible score to be obtained from the respective task

maximumValue a number value describing the highest possible score to be obtained from the respective task

valueType a string value describing the form of the assessment; can be either 'NUMBER' or 'STRING' or 'BOOLEAN'

identifier a string value representing the identifier of the definition

hidden a boolean value describing whether the definition is hidden within the hierarchy or not

expression a string value; holds the expressions that can be executed upon the definition(s)

evaluationType a string value describing the way of processing the respective classifications; 'MANUAL' or 'EXPRESSION' is supported

courseCode a string value containing the code of a course the definition belongs to

classificationType a string value; describes the type of the task corresponding to the definition; ex. 'HOMEWORK', 'ACTIVITY', 'QUIZ', 'ASSESSMENT' and more

DefinitionWithChildrenInterface The interface extends the already mentioned `DefinitionInterface` interface by adding a `children` property that holds `Definition` instances representing the child definitions. For more information, see figure 3.3.

StudentInterface The `StudentInterface` interface describes the structure of an object representing a student with their assessments using the following properties. More details are presented in figure 3.3.

classifications an object containing `Classification` instances indexed by the value of the respective `definitionIdentifier` property

username a string value; represents the student username

firstName a string value; holds the first name of the student

lastName a string value; holds the last name of the student

email a string value; holds the contact email of the student

personalNumber a string value; holds the personal identification number of the student

StudentGroup The `StudentGroup` interface describes the structure of the data stored about a given student group using the following properties. More details are presented in figure 3.3.

studentGroupId a string value; holds the identifier of the student group

name a string value; holds the name of the student group

description a string value; holds additional information about the group

type a string value; holds information about the type of the student group; for example, 'LECTURE'

ClassificationInterface The `ClassificationInterface` interface describes the data stored about a single student assessment; specifically:

definitionIdentifier a string value; represents the respective task

studentUsername a string value; holds the respective student username

value a string, number, or a boolean value representing the student score

note a string value or null; holds text note

timestamp a string value or null; contains the time of submission

For more detail regarding the interface and other internal and unlisted structures, see figures 3.3 and 3.4.

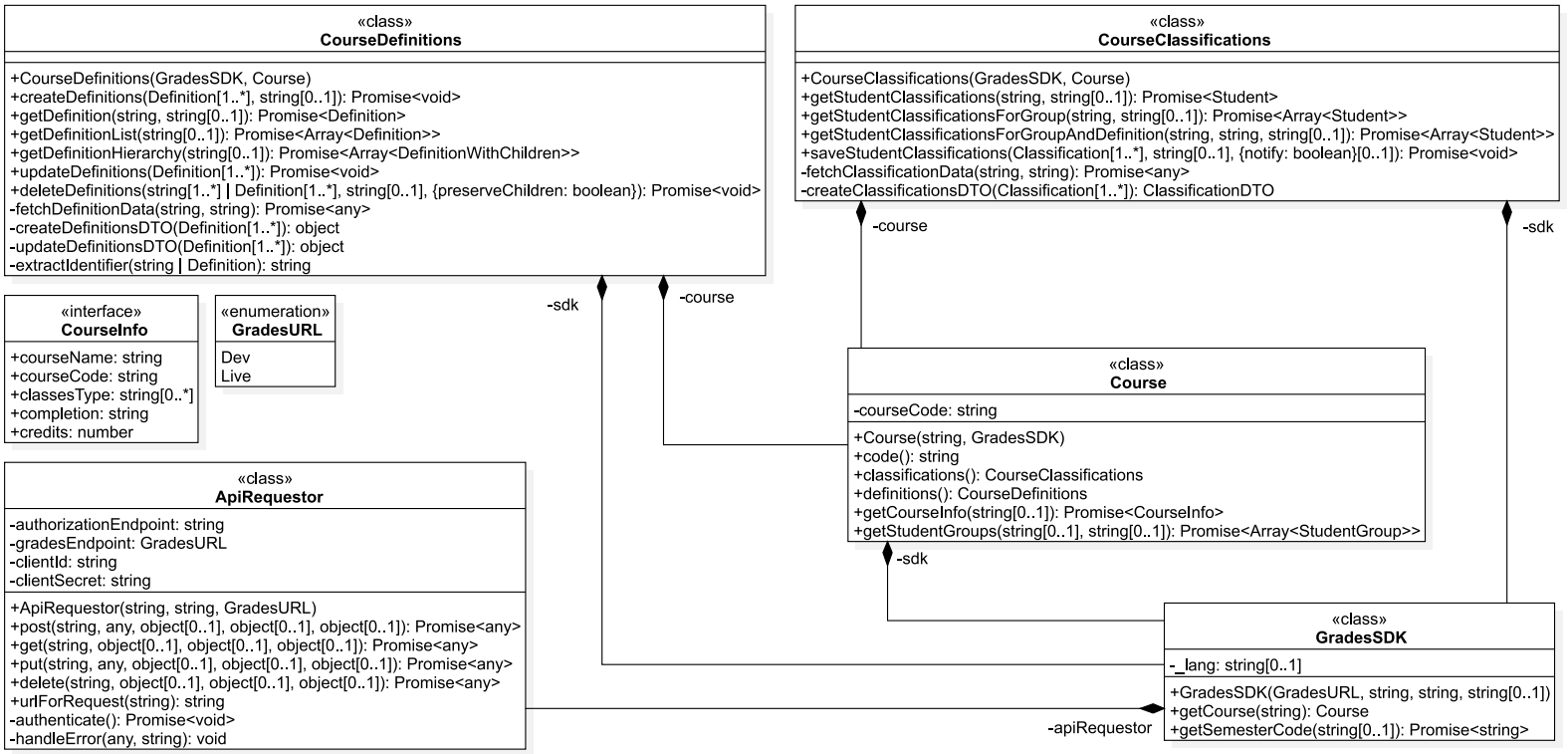


Figure 3.2: Class diagram of the top-level blocks of the module

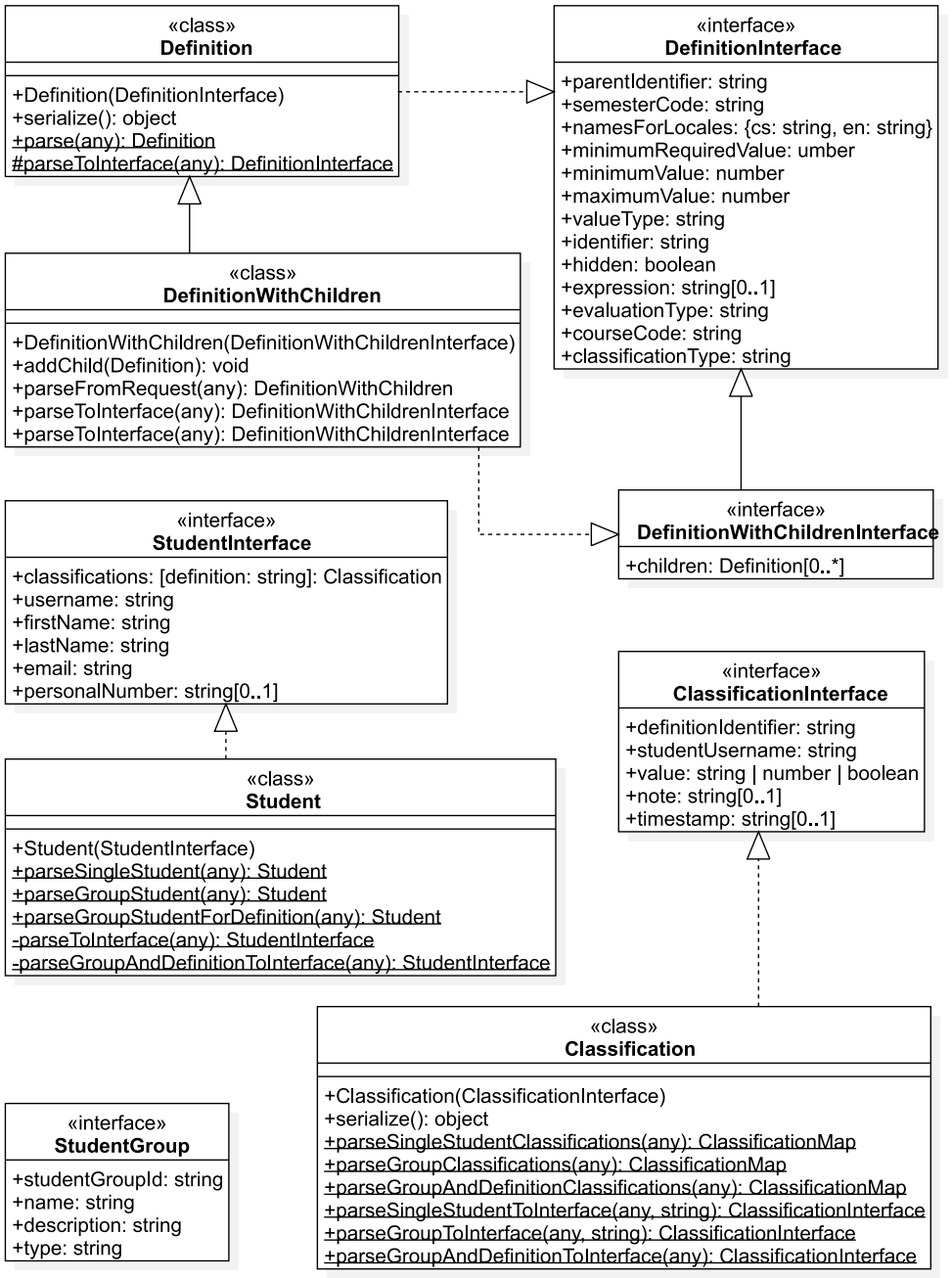


Figure 3.3: Class diagram of the middle-level blocks of the module

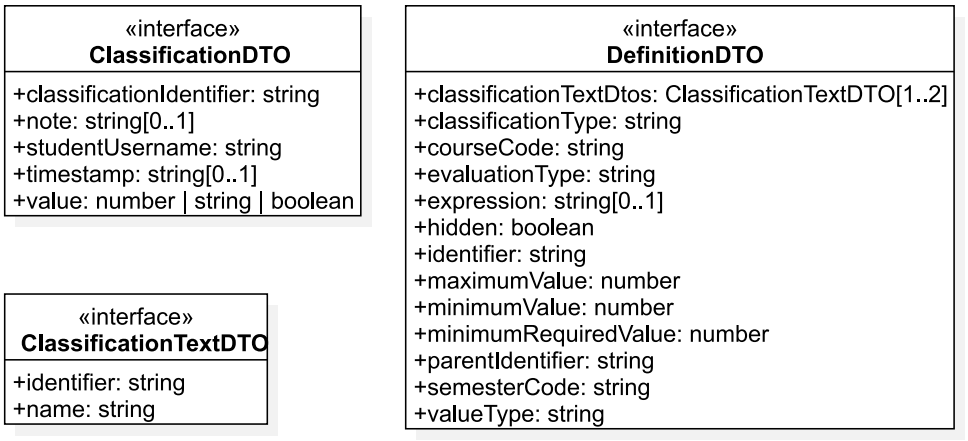


Figure 3.4: Diagram of the DTO (data transfer object) interfaces

Chapter 4

Implementation

This chapter describes the npm module implementation process. The module is implemented in TypeScript and published as an NPM module. In order to integrate the module into the system of university applications and obtain access to Grades API, registration into the university Apps Manager was required. The module has several developer dependencies; however, the built distributable only relies on one dependency – Axios. Furthermore, the module functionality is demonstrated using a custom proof-of-concept application.

4.1 Module

The following sections describe the implementation process, including setting up the project, its Git repository, and TypeScript dependency.

4.1.1 Project Setup

After the initial installation of the Node.js framework using the appropriate installer[4], an empty npm project was created using the command `npm init -y`. The command created a `package.json` file storing the module's metadata, including the name, description, and version number, as shown in listing 1.

4.1.2 Git Repository & User Documentation

A Git repository needed to be created for code management and distribution; therefore, `README.md` and `.gitignore` files were added to the project. The `README.md` file serves as a user guide, providing the setup and installation instructions and introducing module functionality and usage. The `.gitignore` file contains paths to files and directories that Git should ignore; this includes

[4]available from https://nodejs.org/en/download/

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {"test": "echo \"Error: no test spec.\" && exit 1"},
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Listing 1: The package.json file of the empty npm project

any build data, the `node_modules` directory (contains the installed dependencies), any files and folders containing test reports, and finally, the hidden files and directories created by the development environment or the OS. A repository tag was added into the `package.json` file, as shown in listing 2, to connect the Git repository with the module project.

```
"repository": {
    "type": "git",
    "url": "https://gitlab.fit.cvut.cz/hampejar/grades-sdk"
}
```

Listing 2: Connecting the project with a Git repository

4.1.3 Setting up TypeScript

TypeScript is used as the primary development language. The usage of TypeScript requires adding TypeScript as a developer dependency to the project. Dependencies can be easily added/installed into an existing project using the `npm install "package-name"` command. The TypeScript dependency is only relevant during the development since the distributed version of the module is transpiled into JavaScript. Adding an argument `--save-dev` to the install command ensures the dependency will not be included in the published module. After installing TypeScript by using the `npm install --save-dev typescript` command, it is necessary to add a `tsconfig.json` file. This file includes the information needed for the TypeScript transpiler. The final structure and contents of the file can be seen in listing 3.

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "declaration": true,
    "outDir": "./lib"
  },
  "include": ["src"],
  "exclude": ["node_modules", "tests"]
}
```

Listing 3: Structure and contents of the tsconfig.json file

4.1.4 Authentication

At the beginning of the module implementation, a way of authenticating the module needed to be resolved to gain full access to Grades API. It was first necessary to create a project in the Apps Manager (seen in figure 4.1) managing university-level application access to CTU APIs. After creating the project, an identifier and a secret are assigned to the project. Access to the necessary services can be set up in the *Services* tab of the Apps Manager, shown in figure 4.2. For this project, the Courses service was relevant. The Course service offers several scope levels for working with Grades API. For the developed module, the `cvut:grades:course-restricted` scope was sufficient. The assigned identifier and secret can later be used to obtain the final access token necessary for authentication with Grades API.

4.1.5 Implementation

The implementation of the module functional aspects followed after the initial setup. The structure of the implemented module corresponds to the design presented in the previous chapter. A `src` directory was created to store the source code. As seen in the configuration file in listing 3, only the `src` directory is visible to the transpiler and will be present in the published module in the transpiled form. The project can be built by running the command `npm run build`. The built files occupy the `lib` directory, as specified in the `tsconfig.json` file.

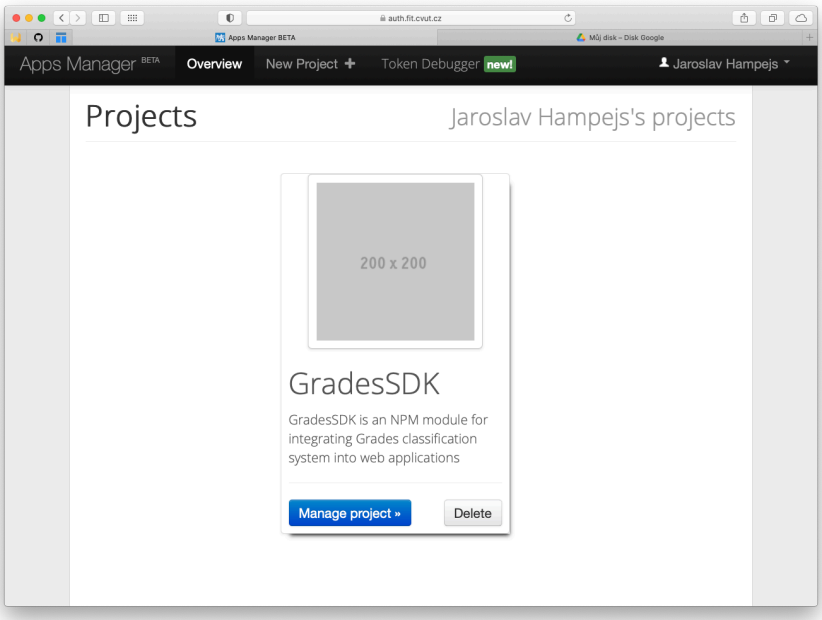


Figure 4.1: GradesSDK project in the Apps Manager

4.2 Dependencies

The implemented module depends on a single external package, the Axios package managing the HTTP requests and responses. The `ApiRequestor` class implements communication with Grades API and uses Axios. Several additional dependencies were also utilized during the development and are not present in the released module:

typescript The `typescript` package was used to bring typescript features into the development, mainly type support, allowing greater control over the data flow inside the module.

dotenv The `dotenv` package allows for storing and loading environment variables. It is utilized in the PoC application for storing the clientID and clientSecret variables used for acquiring the access token necessary to access the full functionality of Grades API.

jest, ts-jest and @types/jest Jest packages were used in the testing stages of the development for UNIT testing the developed module.

typedoc The `typedoc` package was used to generate the implementation documentation of the final module. This documentation is attached in the appendix and can also be found on the GitLab page of the module.

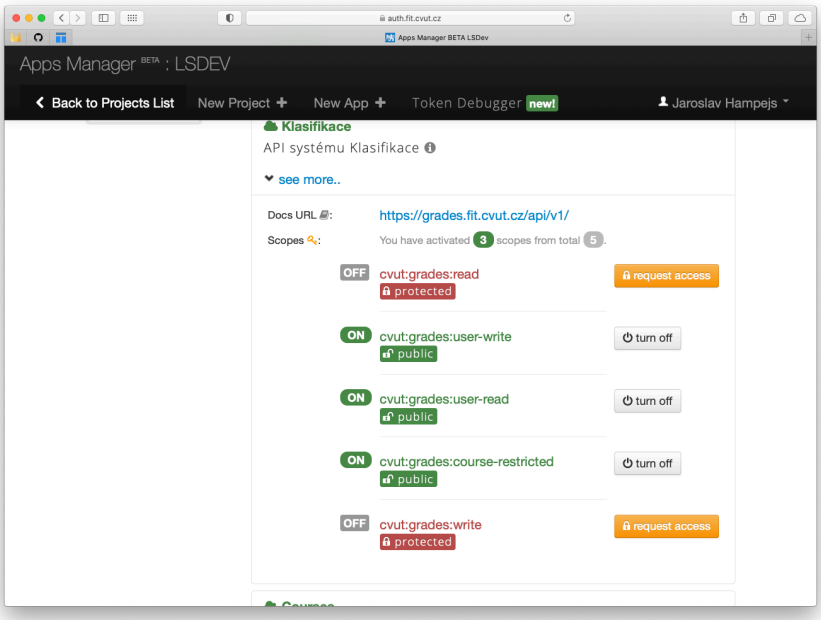


Figure 4.2: Services tab in the Apps Manager

4.3 Proof-of-Concept application

A proof-of-concept Node.js application was developed to check the functionality and usability of the implemented module. The application showcases the features equivalent to the functional requirements presented in section 2.5. The features of the PoC application include:

Viewing course metadata As shown in figure A.1, the app can display information about a course in a specified semester.

Managing course definition hierarchy The figure 4.3 shows the app presenting the definition hierarchy, figure A.2 shows a form for adding a new definition into the hierarchy. Finally, figure 4.4 shows the same definition hierarchy via Grades.

Managing student assessments The PoC app also allows managing student assessments as shown in figure 4.5; the same data viewed via Grades can be seen in figure A.3. Displaying assessment details, including notes, is also possible, as shown in figure A.4.

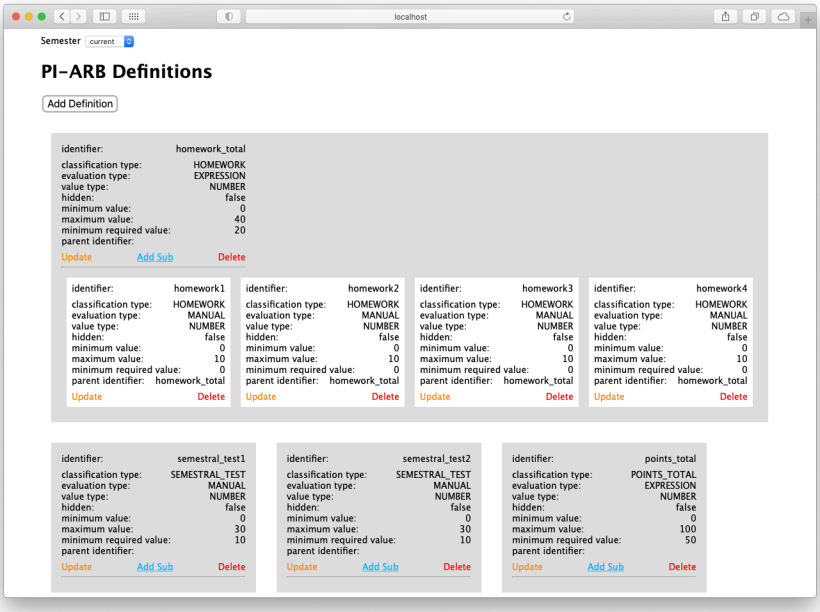


Figure 4.3: Viewing course's definition hierarchy in the PoC application

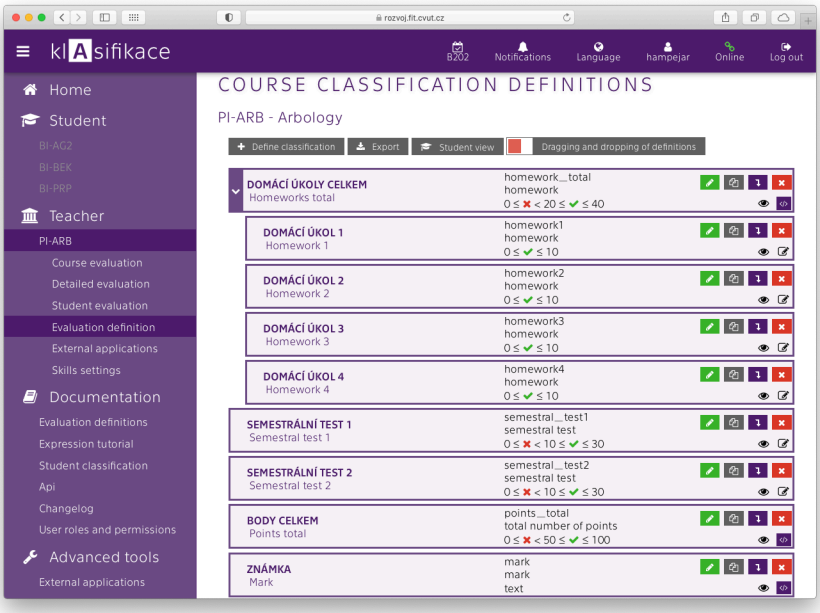


Figure 4.4: Viewing course's definition hierarchy via Grades web interface

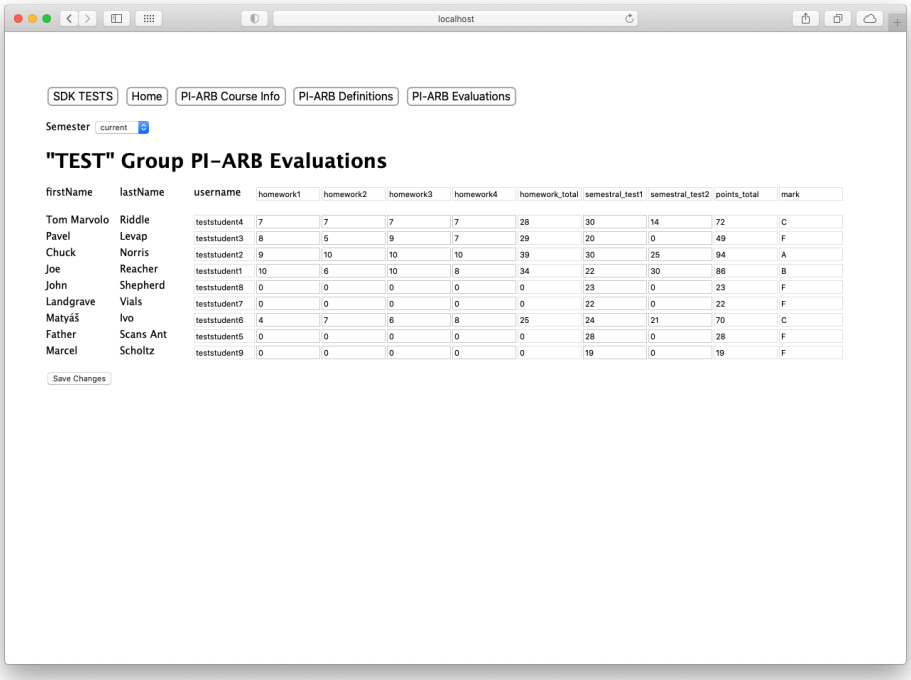


Figure 4.5: Viewing student's evaluations in the PoC application

Chapter 5

Testing and Distribution

This chapter describes the module testing process and touches on setting up the testing software and its subsequent use. Furthermore, this chapter presents the way of publishing the finalized module to the NPM registry.

5.1 Testing the Module

To test the module functionality, a JavaScript unit testing framework, *Jest*, is used. TypeScript support for the tests is added by installing additional developer dependencies for the *Jest* framework, including `@types/jest` and `ts-jest`.

The main objective of unit testing is to test the individual parts/units of the software. It is essential to avoid testing the units that are not part of the developed software. Therefore, for testing purposes, it is first necessary to substitute the Grades API. In order to shield off the networking, the necessary functionality of the Axios package is mocked, i.e., temporarily reimplemented locally, which includes Axios' `get`, `post`, `put`, and `delete` methods.

A directory titled `tests` contains files required for testing. The configuration file `jestconfig.json` stores settings for the *Jest* framework. A total of nineteen tests, including the following ones, cover all of the functionality of the module:

Should get semester code Tests whether or not the module returns the correct codes for the previous, current, and following semesters.

Should get course information Tests whether or not the module returns well-formatted course metadata.

Should get definition hierarchy Tests whether or not the module returns a well-formatted object representing the definition hierarchy of a given course.

Should delete definition Tests whether or not the module attempts to delete a specified definition from the course's definition hierarchy.

Should get all definitions Tests whether or not the module returns a flattened array containing definitions from the course's definition hierarchy.

Should create a definition Tests whether or not the module attempts to create and add a new definition to the course's definition hierarchy.

Should save multiple student classifications Tests whether or not the module handles creating/updating multiple classifications at once.

Should get student groups Tests whether or not the module returns the student groups of a given course.

Should throw an authentication error Tests whether or not the module responds as defined to an authentication error.

All tests are passing. The final report with included coverage results can be seen in figure 5.1.

5.2 Publishing the Module

After pushing the latest changes of the source code to the Git repository (including updating the version number of the module in the `package.json` file), the module is published by running the `npm publish` command. The same command with a `--dry-run` argument can be executed beforehand to preview the list of files to be published. The npm page of the module is available at `https://www.npmjs.com/package/grades-sdk`.

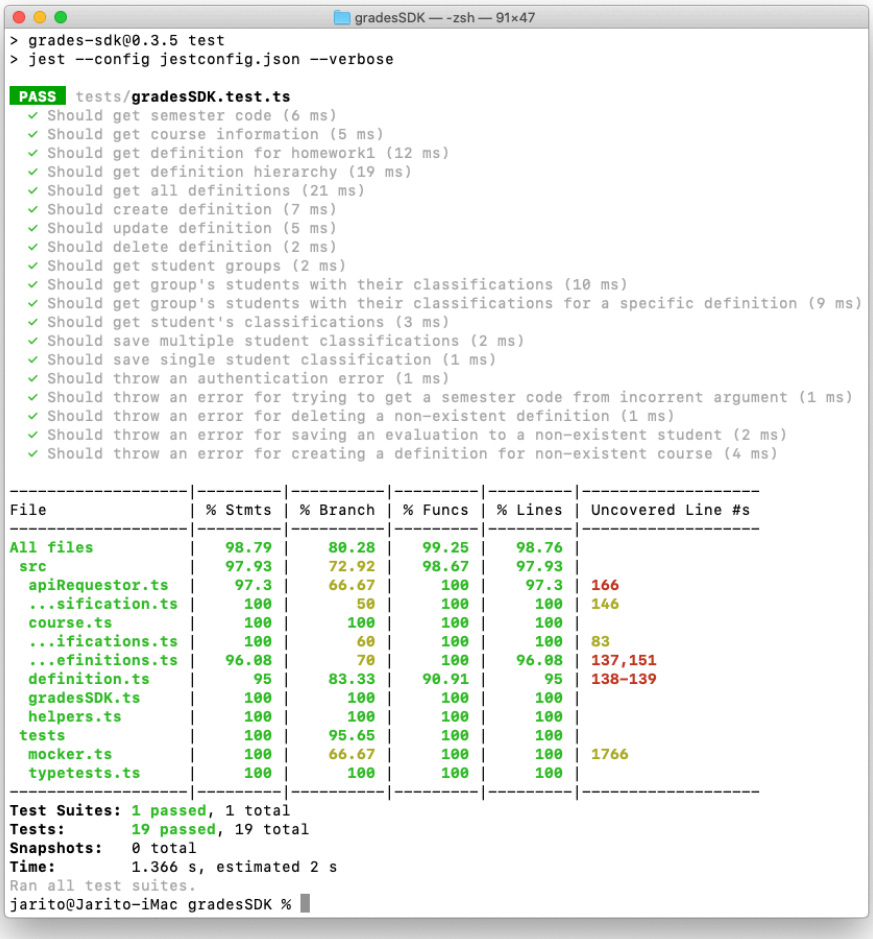


Figure 5.1: Jest testing report with coverage for the module

Conclusion

The main goal of this thesis was to provide a way to connect independent educational platforms with Grades. It was necessary to analyze the current state of educational web applications and services at FIT CTU and design and implement a tool that would allow simple integration of the Grades assessment system into existing and future educational applications, including LearnShell.

The research focused on the current state of electronic assessment at FIT CTU, the Grades and LearnShell applications, modular programming, REST and REST APIs, and other relevant web technologies.

In the analysis, the necessary functionalities of the Grades synchronization tool were defined. The selection of appropriate technologies for the development and possibilities of code distribution were also discussed. To ensure reusability, ease of use, and simple integration with present educational apps, the design of the proposed tool uses an object-oriented interface and is implemented as a TypeScript NPM module leveraging the Grades REST API.

The module can be easily integrated into any JavaScript or TypeScript application or service in the Node.js environment. The module was published to the NPM registry and is freely available. Unit tests covering over 95% of the module's functionality and internal data management are available. Concurrently, a proof-of-concept application was also developed to test and showcase the module's interface functionality and usability. The module can be easily installed into the LearnShell project by following the directions provided on the module's NPM or GitHub web pages.

The module does not support several functionalities provided by Grades at this stage since they were not considered to be crucial for essential course management. Adding support for these segments of the Grades API is subject to further development. The proof-of-concept application and unit testing results show that all the requirements defined in the thesis assignment, introduction, and analysis have been met.

Bibliography

[1] FIT CTU. *FIT Classification REST API* [online]. [cit. 2021-04-15]. Available from: https://grades.fit.cvut.cz/api/v1/swagger-ui.html

[2] GUTH, O. RESTful web services. In: *FIT CTU Courses* [online], 2020, [cit. 2020-11-25]. Available from: https://courses.fit.cvut.cz/BI-TJV/lectures/files/bi-tjv-p-rest.pdf

[3] CLARKSON, M. R. *Functional Programming in OCaml* [online]. 2019, [cit. 2021-04-20]. Available from: https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/modules/modular_programming.html

[4] blender.org. *Modules* [online]. 2021, [cit. 2021-04-20]. Available from: https://wiki.blender.org/wiki/Modules

[5] Create, read, update and delete. In: *Wikipedia, The Free Encyclopedia* [online], 2020, [cit. 2020-11-27]. Available from: https://en.wikipedia.org/w/index.php?title=Create,_read,_update_and_delete&oldid=990607204

[6] WILDERMUTH, S. *REST matters (and you need more of it)* [online]. *Pluralsight*, 2015, [cit. 2020-11-26]. Available from: https://www.pluralsight.com/blog/tutorials/representational-state-transfer-tips

[7] RFC 2616. *Hypertext Transfer Protocol — HTTP/1.1* [online]. W3C, 2004, rev. 1.8. [cit. 2020-11-26]. Available from: https://www.w3.org/Protocols/rfc2616/rfc2616.html

[8] Tutorialspoint. *HTTP–Requests* [online]. 2020, [cit. 2020-11-30]. Available from: https://www.tutorialspoint.com/http/http_requests.htm

[9] BLOOMER, J. *Power Programming with RPC* [online]. Nutshell handbook, O'Reilly Media, Incorporated, 1992, ISBN 9780937175774, [cit. 2021-04-21]. Available from: https://books.google.cz/books?id=PN2hcRD29JUC

[10] STURGEON, P. Understanding RPC Vs REST For HTTP APIs. *Smashing Magazine* [online], 2016, [cit. 2020-11-27]. Available from: https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/

[11] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation thesis, University of California, Irvine, 2000, [cit. 2021-04-21]. Available from: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[12] MDN contributors. *Synchronous and asynchronous requests* [online]. 2020, [cit. 2021-04-18]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

[13] MDN contributors. *XMLHttpRequest* [online]. 2021, [cit. 2021-04-18]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

[14] BANGARE, S. *The Fetch API — A modern replacement for XMLHttpRequest* [online]. 2018, [cit. 2021-04-18]. Available from: https://medium.com/beginners-guide-to-mobile-web-development/the-fetch-api-2c962591f5c

[15] MDN contributors. *Using Fetch* [online]. 2021, [cit. 2021-04-18]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[16] JavaScript.Info. *Promise* [online]. 2021, [cit. 2021-04-18]. Available from: https://javascript.info/promise-basics

[17] MDN contributors. *Promise* [online]. 2021, [cit. 2021-04-18]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[18] TANEJA, S.; GUPTA, P. R. Python as a Tool for Web Server Application Development. *JIMS 8i-Int'l J. of Inf. Comm. & Computing Technology(IJICCT)* [online], volume 2, no. 1, 2014: pp. 77–83, [cit. 2021-04-18]. Available from: https://www.jimsindia.org/8i_Journal/VolumeII/Python-as-a-tool-for-web-server-application-development.pdf

[19] Django Software Foundation. *Models* [online]. 2020, [cit. 2020-11-28]. Available from: https://docs.djangoproject.com/en/3.1/topics/db/models/

[20] Django Software Foundation. *Templates* [online]. 2020, [cit. 2020-11-28]. Available from: https://docs.djangoproject.com/en/3.1/topics/templates/

[21] Django Software Foundation. *Class-based views* [online]. 2020, [cit. 2020-11-28]. Available from: https://docs.djangoproject.com/en/3.1/topics/class-based-views/

[22] GEORGE, N. *A slightly different view of Django's MTV "stack"* [online]. 2020, [cit. 2020-11-27]. Available from: https://djangobook.com/wp-content/uploads/mtv_drawing1_new.png

[23] GEORGE, N. *A more holistic view of Django's architecture* [online]. 2020, [cit. 2020-11-27]. Available from: https://djangobook.com/wp-content/uploads/mtv_drawing2_new.png

[24] JOHANSSON, A. *5 Reasons JavaScript Is Still Better Than Python* [online]. 2021, [cit. 2021-04-18]. Available from: https://www.computer.org/publications/tech-news/build-your-career/5-reasons-javascript-is-still-better-than-python

[25] TypeScript. In: *Wikipedia, The Free Encyclopedia* [online], 2021, [cit. 2021-04-18]. Available from: https://en.wikipedia.org/w/index.php?title=TypeScript&oldid=1017626046

[26] npm, Inc. *Documentation for the npm registry, website, and command-line interface* [online]. [cit. 2021-04-19]. Available from: https://docs.npmjs.com

[27] Stack Overflow. *Developer Survey Results 2019* [online]. 2019, [cit. 2021-04-16]. Available from: https://insights.stackoverflow.com/survey/2019#technology-_-most-loved-dreaded-and-wanted-web-frameworks

[28] Vercel, Inc. *Create a Next.js App* [online]. 2020, [cit. 2020-11-28]. Available from: https://nextjs.org/learn/basics/create-nextjs-app

[29] KONSHIN, K. *Next.js Quick Start Guide: Server-side rendering done right* [online]. Packt Publishing Ltd, 2018, ISBN 978-1-78899-366-1, [cit. 28. 11. 2020]. Available from: https://books.google.cz/books?hl=cs&lr=&id=-rBmDwAAQBAJ&oi=fnd&pg=PP1&dq=next.js&ots=xdBNSxyRFh&sig=3wg_Wa_G61Ll1r3yFKvK7h8jO6k&redir_esc=y#v=onepage&q=next.js&f=false

Appendix A

Attachments

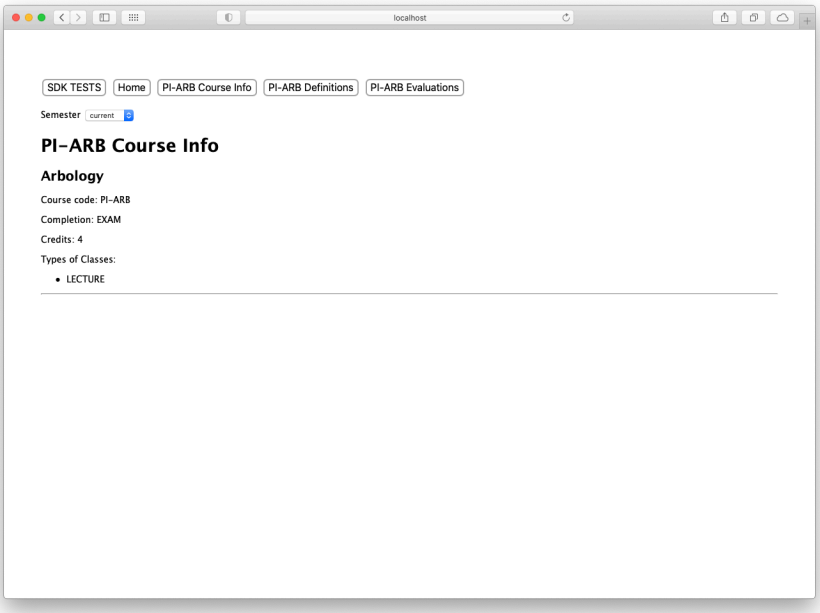


Figure A.1: Viewing course metadata in the PoC application

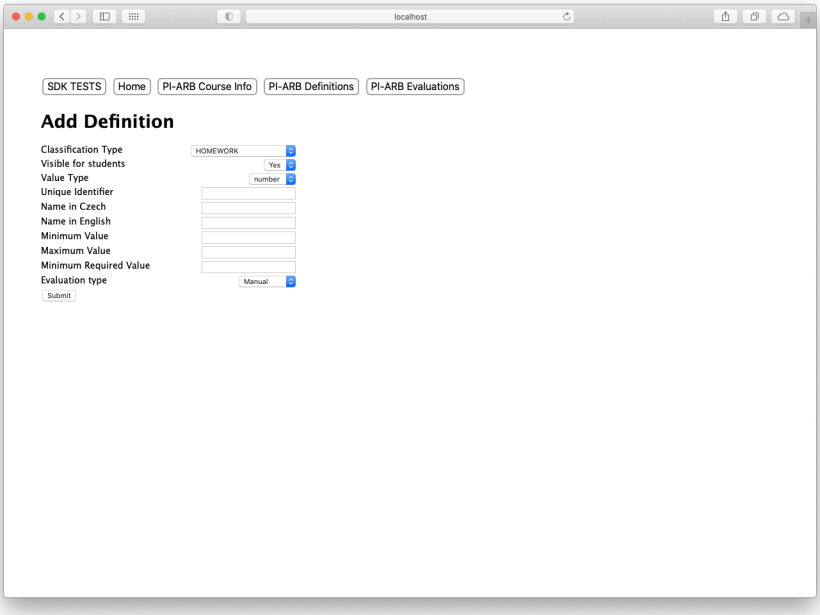


Figure A.2: The form for adding a definition into the hierarchy

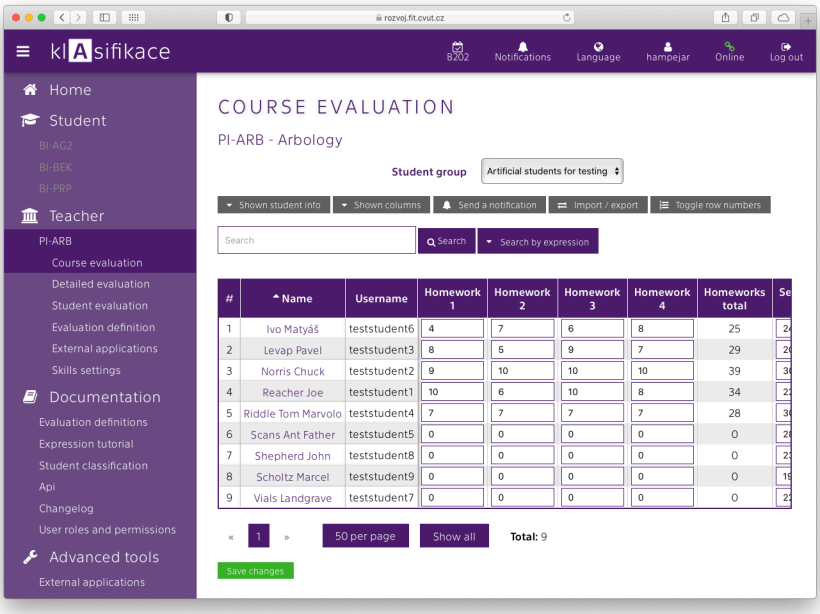


Figure A.3: Viewing student group's classifications via Grades web interface

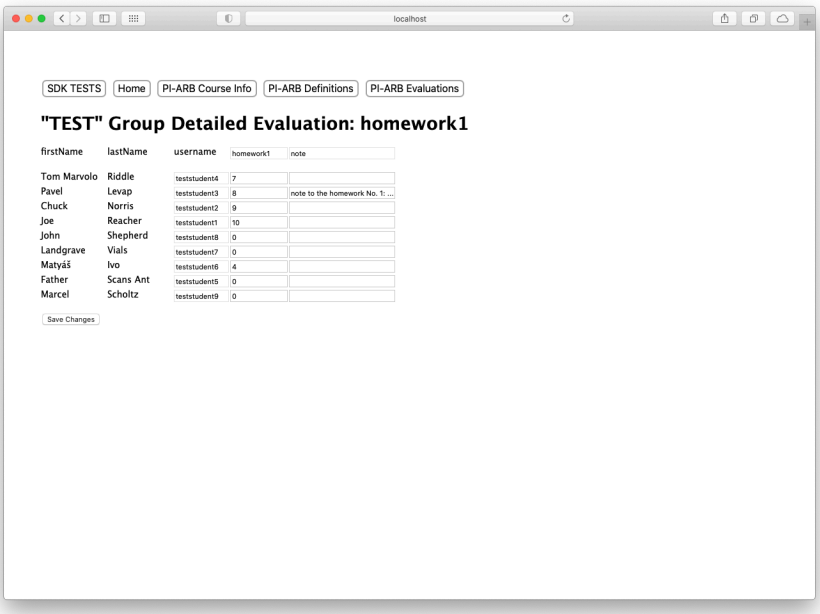


Figure A.4: Viewing detailed evaluations in the PoC application

Appendix B

Acronyms

SDK Software Development Kit

API Application Programming Interface

NPM Node.js Package Manager

REST Representational State Transfer

JS JavaScript

TS TypeScript

HTML Hypertext Markup Language

CLI Command Line Interface

PoC Proof of Concept

XHR XMLHttpRequest

SEO Search Engine Optimization

CSS Cascading Style Sheets

URL Uniform Resource Locator

SOAP Simple Object Access Protocol

RPC Remote Procedure Call

OS Operating System

Appendix C

Contents of Enclosed CD

- `README.txt` ..... CD content README file
- `BP_Hampejs_Jaroslav_2021.pdf` ..... thesis in PDF format
- `gradesSDK` ..... module project directory
  - `coverage` ..... test coverage directory
  - `docs` ..... implementation documentation directory
  - `lib` ..... build files directory
  - `src` ..... source code directory
  - `tests` ..... unit tests directory
  - `.env.template` ..... template for the environment variables file
  - `jestconfig.json` ..... unit test configuration file
  - `LICENSE` ..... license description file
  - `package.json` ..... dependency definition file
  - `package-lock.json` ..... dependency tree file
  - `tsconfig.json` ..... configuration file for the TS transpiler
  - `README.md` ..... user manual
- `ls-grading` ..... test app project directory
  - `bin` ..... directory for startup scripts
  - `public` ..... UI files directory
  - `routes` ..... route definitions directory
  - `views` ..... HTML page templates directory
  - `.env.template` ..... template for the environment variables file
  - `app.js` ..... application entry point and set-up file
  - `package.json` ..... dependency definition file
  - `package-lock.json` ..... dependency tree file
  - `README.md` ..... installation manual