Grading Ecosystem – Architecture



Contents

[Introduction 4](#_Toc370550667)

[High-Level Design 5](#_Toc370550668)

[General Overview 5](#_Toc370550669)

[Data Model 7](#_Toc370550670)

[Web Services 10](#_Toc370550671)

[Technologies 14](#_Toc370550672)

[Low-level Design 15](#_Toc370550673)

[Shared Part 15](#_Toc370550674)

[Development View 15](#_Toc370550675)

[Logical View 15](#_Toc370550676)

[Server 15](#_Toc370550677)

[Development View 15](#_Toc370550678)

[Logical View 15](#_Toc370550679)

[Process View 15](#_Toc370550680)

[Physical View 15](#_Toc370550681)

[Database model 15](#_Toc370550682)

[Admin Web Application 15](#_Toc370550683)

[Integrations 15](#_Toc370550684)

[Spoj0 Integration 15](#_Toc370550685)

[Arena Maycamp Integration 16](#_Toc370550686)

[Eclipse Plug-In 16](#_Toc370550687)

[Performance Testing 16](#_Toc370550688)

[Security 16](#_Toc370550689)

[Deployment 16](#_Toc370550690)

[Future Considerations 16](#_Toc370550691)

[Server wish-list 16](#_Toc370550692)

[Visual Studio Add-On 16](#_Toc370550693)

[Advanced Grader 17](#_Toc370550694)

[References 17](#_Toc370550695)

|  |  |  |
| --- | --- | --- |
| **Author** | **Date Changed** | **Note** |
| Martin Toshev | 01.06.2013 | Initial draft |
| Martin Toshev | 19.10.2013 | Revised architecture – changed high level overview diagram and descriptions of components. Added more details to the low level design section. |
|  |  |  |
|  |  |  |
|  |  |  |

# Introduction

The purpose of this document is to describe in details the architecture of a system called **Grading Ecosystem** that provides a number of enhancements and new features to a the set of existing grading systems used for both university and school educational purposes in the field of Computer Science. These enhancements and features include:

* unified data model for the various artifacts used by the system (contests, problems);
* easier submission of problems using an administrative web application;
* easier administration and maintenance;
* integration of different grading systems.

Furthermore there are several quality attributes (ordered by priority) considered by the architecture:

* performance – the system should be able to undertake a load of a relative large number of simultaneous users (e.g. 400-500 hundred) that perform intensive tasks;
* security – the system should minimize security concerns regarding the particular user (whether this is a teacher, contestant or administrator);
* scalability – this system should be able to scale both horizontally and vertically, although this is not a strict requirement – it is a necessary in the case when performance should be preserved in a growing community of users of a single deployment. Load balancing is additionally planned;
* availability – since it is important that the system should be highly available (especially when doing an upgrade or there is a peak load during, for example, a contest) - load balancing and failover are considered;
* fault tolerance – the system should perform an adequate level of error handling so that the system does not go down in case of a user-triggered or system fault;

The system is pluggable and allows for integration with various third-party clients and multiple grading systems. In essence – it is a middleware for grading systems.

# High-Level Design

## General Overview

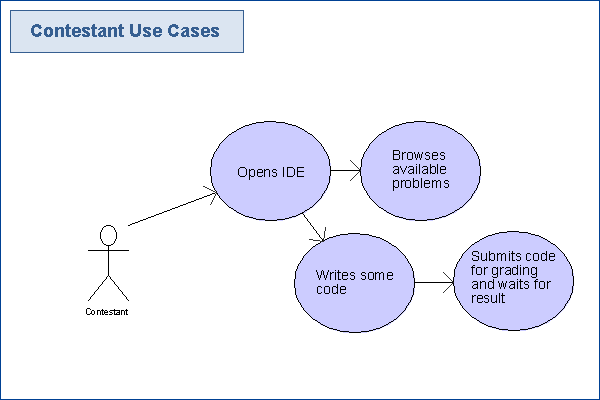
The following diagram gives an overview of the system:

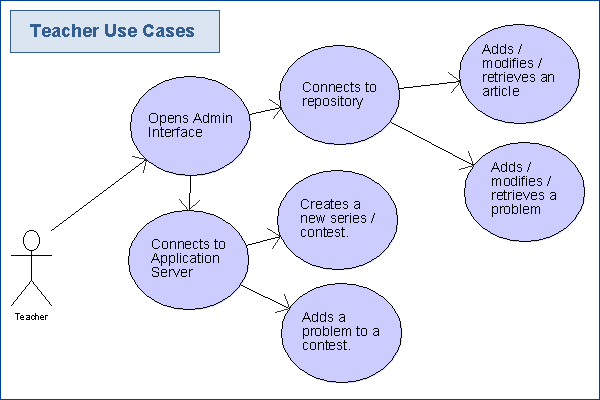


At the core of the system there is an application server and central repository for storing contest data:

* The repository provides a set of RESTful web services for CRUD operations on problems. It is intended to serve as a central repository for problem data that is manageable from the Admin Interface – problem data from the repository can be used to create supply contest data to the particular grading systems and the particular grading systems might be used to supply problems to the repository;
* The application server provides a set of RESTful web services for performing CRUD operations on a series of contests (here a series could be a particular course, for example), a particular contest in a series, or a particular problem in a contest (along with the test data) and for submission of problem solutions in the form of source code. All of these services are targeted at a particular grading system integrated into the application server by means of a mechanism for supplying adapters for the different grading systems. Third party applications targeted for contestants (such as the various IDE integration provided to contestants) connect to the application server for problem retrieval and submission. Teachers are able to create series, contests and add problems to contests via the administrative interface.

The following diagrams display the typical use cases for the different types of users:





We will further describe in details the low-level details of the particular components using the 4+1 architectural view model in a subsequent section.

## Data Model

Before we can give a more detailed overview on the system internals the data model used by the application will be outlined. We have used the CORE (Contests Repository) model outlined in [2] with modifications and extensions provided by the model outlined in [5]. Since the outlined data is presented using JSON format (JavaScript Object Notation) we will be using the same format. This format will also serve as a basis for describing later the schema-less structure of the data as persisted in a Mongo database.

We distinguish between the following data types:

**Series** – defines a grouping of contests and problems  
Example:

{

"id": "<series\_id>",

"title": "Design and analyses of algorithms - 2013",

"about": " Lectors :... , Hours :... etc .",

"notes": "( Some sensitive information )"

"contest\_order" : [ "homework 1"]

"problem\_order" : [ "problem 1"]

"parent": "Courses 2013"

}

**Contest** – defines a programming contest that has a set of problems  
Example:

{

"id": "<contest\_id>",

"type": "standard",

"title": "Homework 1",

"start\_time": "2012-06-28 09:00:00",

"duration": 300,

"about": "",

"grading\_style": "acm",

"problem\_order ": ["fish", "honey", "swim"],

"problem\_scores": [120 , 150 , 80]

}

**Problem** – defines a particular problem  
Example:

{

"id": "<problem\_id>",

"type": "standard",

"title": "hw1 - problem -A",

"description": "... problem description ...",

"time\_limit": 4.5,

"memory\_limit": 64,

"origin": "Taken from Contest -X",

"categories": ["Dynamic Programming"],

"authors":["author1", "author2"],

}

**Test** – defines a particular problem  
Example:

{

"id": "<test\_id>",

"problemId": "<problem\_id>",

"weight": 10,

"content": "1 218 6 7",

}

**Author Solution** – defines an author solution for a particular problem  
Example:

{

"id": "<author\_solution\_id>",

"problemId": "<problem\_id>",

"source": "<source\_code>",

"lang": "<source\_language>"

}

**Checker** – defines a checker for a particular problem  
Example:

{

"id": "<test\_id>",

"problemId": "<problem\_id>",

"source": "<source\_code>",

"lang": "<source\_language>",

"binary": "<checker\_binary>"

}

**Category** – defines the programing category of a problem/article  
Example:

{

"id": "<problem\_category\_id>",

"name": "Spanning Trees",

"description": "Problems on spanning trees",

"parent": "Graph Theory"

}

**Grader Instance** – defines a grader instance  
Example:

{

"id": "<grader\_instance\_id>",

"type": "spoj0",

"name": "spoj0-1",

"URL": "http://judge.openfmi.net:9080/spoj0/" ,

"administration\_enabled" : "true"

}

**Submission** – defines a problem submission from a contestant  
Example:

{

"id": "<submission\_id>",

"user": {"handle" : "martin", "pass" : "martin\_pass"},

"grader\_instance\_id": "spoj0\_1"

"series": "Design and analyses of algorithms - 2013",

"contest": "Homework 1",

"problem": "hw1 - problem -A",

"source": "... source code ...",

"language" : "Java",

"status": "ok",

"log": ""

}

**Grader User** – defines a user handle for a particular grader  
Example:

{

"id" : "<grader\_user\_id>",

"handle" : "martin\_spoj0",

"pass": "skajdUOdsa77sadKL"

}

**User** – defines a user of the system  
Example:

{

"id" : "<user\_id>",

"handle" : "martin",

"name" : "Martin Toshev",

"pass" : "cX6kajsdU76",

"role" : "contestant",

"details" : "",

"permissions" : "rw"

}

**Article** – defines an article (e.g. a tutorial entry) for a particular algorithm, data structure or problem

Example:

{

"id" : "<article\_id>",

"title" : "Dijkstra’s Algorithm",

"format" : "pdf",

"content" : "…",

"authors" : ["Martin Toshev", "Orlin Tenchev"],

"categories" : ["Graph Algorithms"],

"visible" : "true",

"ref" : "<url\_of\_external\_article>"

}

**Tutorial** – defines a tutorial – a sequence of articles and problems (either global or per-series)

Example:

{

"id" : "<tutorial\_id>",

"name" : "Graph Algorithms",

"content" : [{"type": "article", "ref":"<article\_id>"}, ""]

}

**Tutorial** **Entry** – defines an entry in a tutorial (article or problem)

Example:

{

"type" : "problem",

"ref" : "Problem 1",

"grader\_instance\_id" : "spoj0\_1"

}

## Web Services

The following section defines the web service interfaces used by the application (RESTful web services for the repository and the application server).

For the purpose of simplicity request/response payloads are omitted (the format follows the data model presented in the previous section) – typically JSON/XML is used as the format of sending the REST requests.

Services available from the repository:

* Retrieve available top level problem category IDs:

**GET http://<server\_host>:<server\_port>/categories**

* Retrieve available child category IDs:

**GET http://<server\_host>:<server\_port>/categories/{id}/categories**

* Retrieve category:

**GET http://<server\_host>:<server\_port>/categories/{id}**

* Create a category:

**PUT http://<server\_host>:<server\_port>/categories**

* Edit a category:

**POST http://<server\_host>:<server\_port>/categories/{id}**

* Delete a category:

**DELETE http://<server\_host>:<server\_port>/categories/{id}**

* Retrieve all problems in the specified category

**GET** **http://<server\_host>:<server\_port>/categories/{id}/problems**

* Retrieve all articles in the specified category

**GET** **http://<server\_host>:<server\_port>/categories/{id}/articles**

* Retrieve all problems (at least one category must be provided as a query parameter):

**GET http://<server\_host>:<server\_port>/problems**

Additional request parameters:

**type** – problem type used to filters the retrieved problems   
**categories** – comma-separated list of categories used to filter the problems  
**authors** – comma-separated list of authors used to filter the problems

* Retrieve a particular problem:

**GET http://<server\_host>:<server\_port>/problems/{id}**

* Create a problem:

**PUT http://<server\_host>:<server\_port>/problems**

* Edit a problem:

**POST http://<server\_host>:<server\_port>/problems**

* Delete a problem:

**DELETE http://<server\_host>:<server\_port>/problems/{id}**

* Retrieve all articles (at least one category must be provided as a query parameter)

**GET http://<server\_host>:<server\_port>/articles**

Additional request parameters:

**format** – format of the articles  
**categories** – comma-separated list of categories used to filter the articles  
**authors** – comma-separated list of authors used to filter the articles

* Retrieve a particular article:

**GET http://<server\_host>:<server\_port>/articles/{id}**

* Create an article:

**PUT http://<server\_host>:<server\_port>/articles**

* Edit an article:

**POST http://<server\_host>:<server\_port>/articles**

* Delete an article:

**DELETE http://<server\_host>:<server\_port>/articles/{id}**

Services available from the application server:

* Retrieve all available grader types:

**GET http://<server\_host>:<server\_port>/graders/types**

* Retrieve all registered grader instances:

**GET http://<server\_host>:<server\_port>/graders**

* Retrieve a particular grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}**

* Register a grader instance to the server:

**PUT http://<server\_host>:<server\_port>/graders**

* Edit a grader instance:

**POST http://<server\_host>:<server\_port>/graders/{id}**

* Delete a grader instance:

**DELETE http://<server\_host>:<server\_port>/graders/{id}**

* Retrieve all series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/**

* Retrieve child series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}/<series>**

* Retrieve a particular series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}**

* Create a series in a grader in a grader instance:

**PUT http://<server\_host>:<server\_port>/graders/{id}/series**

* Edit a series in a grader instance:

**POST http://<server\_host>:<server\_port>/graders/{id}/series**

* Delete a series in a grader:

**DELETE http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}**

* Retrieve contests from a series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}/contests**

* Retrieve a particular contest from a series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id} /contests/{contest\_id}**

* Create a contest in a series in a grader instance:

**POST http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}/contests**

* Edit a contest in a series in a grader instance:

**PUT http://<server\_host>:<server\_port>/ graders/{id}/series/{series\_id}/contests**

* Delete a contest from a series in a grader instance:

**DELETE http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id} /contests/{contest\_id}**

* Retrieve all problems from a series in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/series/{series\_id}/problems**

* Retrieve all problems from a contest in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/contests/{contest\_id}**

**/problems**

* Retrieve a problem from a contest in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}/contests/{contest\_id}**

**/problems/{problem\_id}**

* Create a problem in a contest in a grader instance:

**PUT http://<server\_host>:<server\_port>/series/{id}/contests/{id}/problems**

* Edit a problem from a contest in a grader instance:

**POST http://<server\_host>:<server\_port>/graders/{grader\_id}**

**/contests/{contest\_id}/problems**

* Delete a problem from a contest in a grader instance:

**DELETE http://<server\_host>:<server\_port>/graders/{grader\_id}**

**/contests/{contest\_id}/problems/{problem\_id}**

* Retrieve submissions for a problem in a grader instance:

**GET http://<server\_host>:<server\_port>/graders/{id}**

**/contests/{contest\_id}/problems/{problem\_id}/submissions**

* Retrieve a submission for a problem:

**GET http://<server\_host>:<server\_port>/graders/{id}**

**/contests/{contest\_id}/problems/{problem\_id}/submissions/{submission\_id}**

* Submit a problem for grading:

**PUT http://<server\_host>:<server\_port>/graders/{id}/submissions**

* Retrieve all tutorials

**GET http://<server\_host>:<server\_port>/tutorials**

* Retrieve a particular tutorial:

**GET http://<server\_host>:<server\_port>/tutorials/{id}**

* Create a tutorial:

**PUT http://<server\_host>:<server\_port>/tutorials**

* Edit a tutorial:

**POST http://<server\_host>:<server\_port>/tutorials**

* Delete a tutorial:

**DELETE http://<server\_host>:<server\_port>/tutorials/{id}**

The following services are available for both the repository and the application server:

* Retrieve all users

**GET http://<server\_host>:<server\_port>/users**

* Retrieve a particular user:

**GET http://<server\_host>:<server\_port>/users/{id}**

* Create a user:

**PUT http://<server\_host>:<server\_port>/users**

* Edit a user:

**POST http://<server\_host>:<server\_port>/users**

* Delete a user:

**DELETE http://<server\_host>:<server\_port>/users/{id}**

Status codes and examples are provided in the client documentation.

## Technologies

The following technologies are used in the project:

* Java SE 7.0 – for the overall implementation of the server application, admin web application and the Eclipse integration;
* Maven – for building the various projects;
* Spring – a DI (dependency injection) container used to provide runtime configuration management for the various projects;
* Apache CXF – web service framework for building SOAP/RESTful web service (provides integration with Spring);
* MongoDB – a NoSQL database (document store) that stored data in the form of JSON documents. All server data (including problem data) is stored in Mongo.
* GIT – a VCS (version-control system) used to store the documentation and source code of the system (a GitHub (see [2]) public repository is used for the purpose).

Additionally the following technologies are used for the various clients:

* Eclipse PDE – for developing of the Eclipse integration;
* Tycho – provides Maven integration for OSGi bundles (used to build the Eclipse-related projects from Maven);
* Visual Studio Add-in API – for developing the Visual Studio integration;
* Apache Velocity – for problem templates in the admin interface;
* iText – for generating PDF documents from the HTML pages of the problems.

# Low-level Design

## Shared Part

The shared part is a separate project that holds common classes and interfaces used by both the server and other applications in the system.

### Development View

### Logical View

## Repository

The repository stores problem data (descriptions, tests for the problems, solutions) and different articles. Typically articles might added internally to the repository from the administrative interface, added from external source and appropriately customized or link entirely to an external source (e.g. an article from TopCoder’s algorithm tutorials [7]). The following diagram provides an overview of the repository.



The repository basically consists of the following components:

* an Apache CXF-based server that provides web services for manipulating the data from the MongoDB database;
* the MongoDB Replica Set that provides the storage engine for the repository data.

The choice of MongoDB as the storage engine is not a coincidence and the idea is derived from [8] where the authors provide a more simplified storage format for contest data in a JSON format that uses the file system.

### Development View

### Logical View

### Database model

## Application Server

The server is the central component of the grading ecosystem and the following sections provide a detailed overview of its architecture. Some of the requirements outlined in [5] apply for the use cases of the system (and in particular – for the server part). The application server provides mechanisms for plugging different grading systems by means of applications with well-defined structure (described later in this section). The following diagram provides an overview of the application server.



### Development View

### Logical View

## Admin Web Application

See document **Grading Ecosystem - Admin Web Application**.

# Integrations

## Spoj0 Integration

See document **Grading Ecosystem - Spoj0 Integration**.

## Arena Maycamp Integration

See document **Grading Ecosystem - Arena Maycamp Integration**.

## Eclipse Plug-In

See document **Grading Ecosystem - Eclipse Plug-In Integration**

# Performance Testing

// describe Apache AB

# Security

// TODO

# Deployment

This section describes the deployment of the system.

// TODO – describe deployment specifics and installer

// TODO – describe the update manager update process (link to Gilad Bracha paper for dynamic classloading)

# Future Considerations

## Server wish-list

Load - balancing using a distributed hash.

Incorporate a tool for plagiarism check.

## Visual Studio Add-On

// TODO

## Advanced Grader

// TODO

# References

[1] A Software Platform for Teaching Programming with Grading Systems,   
K. Manev, M. Sredkov, P. Armyanov  
<http://www.math.bas.bg/smb/2011_PK/tom/pdf/300-305.pdf>

[2] CORE: A Multi-purpose Programming Contest Repository System, Y. Chaushev, M. Sredkov, K. Manev  
<http://www.math.bas.bg/smb/2012_PK/tom_2012/pdf/219-224.pdf>

[3] Design of a Distributed and Asynchronous System for Remote   
Evaluation of Students’ Submissions in Competitive E-learning  
<http://www.ineer.org/Events/ICEE2008/full_papers/full_paper192.pdf>

[4] Assessment of Learner’s Motivation In Web Based E-Learning, N. Ramaha, W, Mohd, F. Ismail  
<http://www.ijser.org/researchpaper%5CAssessment-of-Learners-Motivation-In-Web-Based-E-Learning.pdf>

[5] Software Requirements Specification for ACM Online Judge System, M Orazov, M.Ponamaryov, K. Soral  
<http://www.ceng.metu.edu.tr/~e1527167/docs/srs.pdf>

[6] Automatic Grading of Programming Assignments, MS Thesis, A. Patil  
<http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1050&context=etd_projects>

[7] Topcoder Algorithm Tutorials  
<http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1050&context=etd_projects>

[8] CORE: A Multi-purpose programming contest repository system, Chaushev Y., Sredkov M., Manev K.

<http://sci-gems.math.bas.bg/jspui/handle/10525/1960>

[9] MongoDB Replication  
<http://docs.mongodb.org/manual/replication/>

[10] Grading Ecosystem Project Repository  
<https://github.com/martinfmi/grading_ecosystem>

[11] Apache CXF exception handler for jaxrs (REST)  
<http://www.luckyryan.com/2013/06/15/apache-cxf-exception-handler-for-jaxrs-rest/>