# On-demand virtual laboratory environments for Internetworking e-learning:
# A first step using docker containers

ANDREAS KOKKALIS

January 2018

# Abstract

Learning Management Systems (LMSs) are widely used in higher education to improve the learning, teaching, and administrative tasks for both students and instructors. Such systems enrich the educational experience by integrating a wide range of services, such as on-demand course material and training, thus empowering students to achieve their learning outcomes at their own pace.

Courses in various sub-fields of Computer Science that seek to provide rich electronic learning (e-learning) experience depend on exercise material being offered in the forms of quizzes, programming exercises, laboratories, simulations, etc. Providing hands on experience in courses such as Internetworking could be facilitated by providing laboratory exercises based on virtual machine environments where the student studies the performance of different internet protocols under different conditions (such as different throughput bounds, error rates, and patterns of changes in these conditions). Unfortunately, the integration of such exercises and their tailored virtual environments is not yet very popular in LMSs.

This thesis project investigates the generation of on-demand virtual exercise environments using cloud infrastructures and integration with an LMS to provide a rich e-learning in an Internetworking course. The software deliverable of this project enables instructors to dynamically instantiate virtual laboratories without incurring the overhead of running and maintaining their own physical infrastructure. This sets the foundations for a virtual classroom that can scale in response to higher system utilization during specific periods of the academic calendar.

# Sammanfattning

Add swedish section

# Acknowledgements

I would like to acknowledge ...

# Contents

# List of Figures

# List of Algorithms

# List of Tables

# Listings

# List of Acronyms and Abbreviations

**AJAX**    Asynchronous JavaScript and XML

**API**    application programming interface

**BDD**    Behavior Driven Development

**CA**    Certificate Authority

**CI**    Continuous Integration

**CPU**    Central Processing Unit

**CS**    Computer Science

**CSS**    Cascading Style Sheets

**DOM**    Document Object Model

**DSL**    Domain Specific Language

**EC2**    Elastic Compute Cloud

**e-learning**    electronic learning

**ERB**    Embedded RuBy

**GLUE!**    Group Learning Uniform Environment

**GNU**    GNU's Not Unix!

**GUI**    Graphical User Interface

**HTML**    Hyper Text Markup Language

**HTTP**    Hypertext Transfer Protocol

**HTTPS**    Hypertext Transfer Protocol Secure

**IT**    information technology

**JSON**    JavaScript Object Notation

| | |
|---|---|
| **KTH** | Kungliga Tekniska Högskolan |
| **LIS** | Learning Information Services |
| **LMS** | Learning Management System |
| **LTI** | Learning Tools Interoperability |
| **LTS** | Long Term Support |
| **LXC** | Linux Containers |
| **MIME** | Multipurpose Internet Mail Extensions |
| **MIT** | Massachusetts Institute of Technology |
| **MOOC** | Massive Open Online Course |
| **OCI** | Open Container Initiative |
| **RDBMS** | Relational Database Management System |
| **SCROM** | Sharable Content Object Reference Model |
| **SHA** | Secure Hash Algorithm |
| **SQL** | Structured Query Language |
| **SSH** | Secure Shell |
| **TC** | Tool Consumer |
| **TCP** | Transmission Control Protocol |
| **TLS** | Transport Layer Security |
| **TP** | Tool Provider |
| **TTL** | Time To Live |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **XML** | Extensible Markup Language |

# Chapter 1

# Introduction

The use of electronic learning (e-learning) technologies has been well established in modern education to assist both students and instructors in their learning, teaching, and administrative tasks. One of the e-learning technologies most widely adopted by the academic community is Learning Management Systems (LMSs). A LMS is a software application that handles all aspects of the learning process [2], enabling instructors to design rich e-learning courses and students to experience self-paced learning using a variety of features, such as on-demand course material, video lectures, automatic delivery and evaluation of assignments, collaboration tools, etc.

Many courses, especially in various sub-fields of Computer Science depend on training events in the form of programming assignments, laboratory exercises, simulations, etc. These activities are crucial for students to gain hands-on experience with complex concepts and systems [3]. Although LMSs support on-line training events, such as interactive quizzes with automatic evaluation and analysis of results, providing training events that depend on using complex virtual environments and software are not yet very popular (and hence not widely supported or used).

One of the main advantages of using an LMS is that it supports the integration of external applications to provide personalized, domain specific e-learning, such as messaging and video streaming services, on-line office suites, collaboration tools, or even training environments with exercises tailored to the needs of a specific course.

## 1.1   Background

Hands-on experience is very important to achieve understanding of complex systems and concepts. For example, when studying computer networks, laboratory exercises are a common student activity. An Internetworking course often involves students studying the performance of different Internet protocols under different conditions (such as varying throughput bounds, error rates, and patterns of changes in network conditions).

These experiments depend on specific software, network topologies, and local or virtual hardware. Traditional approaches for realizing such environments depend upon the student's own hardware or on-site computer labs with pre-configured software [4]. More modern approaches involve remote access to virtual machines running on central servers or cloud infrastructures [5].

Currently LMSs do not have built-in support for such laboratory environments. However, one of the main advantages of designing an on-line course on top of an LMS that supports the integration of extenal applications is to provide tailored functionality for the course's and student's specific needs. Today, many LMSs, such as Instructure Inc.'s Canvas [6] LMS, implement the IMS Global Learning Consortium Tools Interoperability® (LTI®) specification. Learning Tools Interoperability (LTI) allows the exchange of information between the LMS and third party components, thus exposing internal functionality of the LMS to external applications in a controlled manner.

Supporting virtual laboratory environments in a LMS in order to meet the needs of an Internetworking course, requires the design of a software framework that implements the LTI interoperability specification in order to exchange relevant information between the laboratory environment and the LMS.

## 1.2 Problem definition

Hands on experience is a key important aspect of the learning process in several fields of Computer Science, including computer networks. Understanding the domain specific concepts and problems of an Internetworking course, depends greatly on exercise material and laboratory practice. Today, such exercises, are not usually designed to extract suitable analytics for the instructor (as an instructor ideally wishes to evaluate each student's level of understanding of each of the different concepts covered in an exercise). Assessing the student's understanding is currently achieved by using additional training material, such as quizzes or assignments in forms of reports which are manually evaluated by instructors or by other students in the form of peer reviews. These alternative methods both introduce a delay in feedback to the student (hence reducing the student's rate of learning) and are not scalable (for example, preventing their use in Massive Open Online Courses (MOOCs)).

Supporting an on-line version of an Internetworking course through a LMS that enables students to achieve the course's learning outcomes at their own pace, depends greatly on designing interactive practice environments. Such environments should be easily modified by the instructor to fit the needs of different exercises. Although today LMSs support a variety of training events, such as quizzes and assignments through integration of external services, on-line virtual laboratory environments that fulfill the requirements of an Internetworking course are not yet well supported and hence not widely used.

However, similar practice environments are common in on-line courses that

teach programming languages. Such environments are part of systems that provide tools for designing coding assignments, and support several assessment methods, including automatic evaluation and grading of code [7] and programming quizzes. These systems often provide standalone web applications or LTI integrations in LMSs that expose functionality for developing code, submitting assignments, and presenting feedback to users [8, 9].

This project aims to design a software framework that supports interactive training material for an Internetworking course, integrates with a LMS to provide a rich e-learning experience, and offers dynamic instantiation of laboratory environments that scale according to the needs of the virtual classroom.

## 1.3 Goals

The design of such a laboratory environment for an Internetworking course has to meet several user requirements from the perspective of both students and instructors, and integrate with a LMS to offer a rich e-learning experience. The expected outcome of this project is a software framework that supports instantiation of on-demand laboratory environments using cloud based technologies to enrich the learning experience of students, allowing them to proceed at their own pace. Additionally, the framework should enable a teacher to customize the environment according to different exercises' requirements, and provide the instructor with constructive feedback about each student's progress and understanding.

The process of designing this framework can be realized by achieving the following goals:

- Devise a method to easily build virtual laboratory environments,

- The framework should enable the instructor to easily create and manage different versions of laboratory environments, as such environments can be reused for different assignments.

- The framework should be integrated with the LMS to enable students to access the training environments via the LMS,

- The method of integration of such exercise environments should be usable by others - thus an important part of this thesis project is documenting the selected method to facilitate the integration of a diverse set of external environments (for example, an ns-3 simulator configured for a particular simulation),

- The framework should scale in such way that it enables students to do assignments at any given time, thus offering on-demand availability of the underlying services, and

- A student should be able to access a training environment within reasonable upper bounded time from the moment she requests from the LMS to start an assignment.

## 1.4 Research Methodology

Design science research addresses important unsolved problems in unique or innovative ways or solved problems in more effective or efficient ways. It focuses on the design and construction of information technology (IT) artifacts that have utility in real-world, application environments. The artifacts, as the outcome of the research process, aim to improve domain-specific systems and processes [10, 11]. The utility, quality, and adequacy of a design artifact, is thoroughly evaluated under varying experimental setups to verify that it successfully fulfills the stated requirements.

Design, in several research fields, including IT, is an iterative process of planning, generating alternatives, and selecting a satisfactory outcome. Design science research, although it is not performed using strictly defined processes, can be summarized by three closely related cycles of activities (these cycles are the relevance cycle, the rigor cycle, and the design cycle) [12], that act as guidelines for designing, constructing, and evaluating an artifact. The relevance cycle establishes the application context that not only provides the requirements for the research as inputs, but also defines acceptance criteria for the evaluation of the research results. The rigor cycle provides past knowledge to the research project to ensure its innovation. It is contingent on researchers to thoroughly research and reference this knowledge base in order to guarantee that the designs produced are research contributions and not routine designs based upon the application of well-known processes. The central Design Cycle iterates between the core activities of building and evaluating the design artifacts and processes of the research [10], until the acceptance criteria, as defined in the Relevance Cycle, are met.

This project is carried out using the design science research approach. The resulting software and documentation attempt to solve the problem of designing and realizing a framework for rich on-line laboratory environments for an e-learning course on Internetworking, that is to be accessible via a specific learning management system (Instructure's Canvas LMS). The two different domains that define the context of this problem are the Internetworking course domain, and the LMS along with the method(s) of integration of external applications into Canvas (in this case via LTI).

## 1.5 Delimitations

This project addresses the problem of designing and integrating virtual laboratory environments to support e-learning in an LMS for an Internetworking course. The

laboratory framework, the expected outcome of this project, has to fulfill several requirements: usability for different types of users (instructor, administrator, and student,), integration into the Canvas LMS via the LTI specification, and satisfy the laboratory and pedagogical challenges of this particular course. Although there are different specifications for integrating external applications and services into a LMS [13], this project addresses only the LTI specifications, as this method is supported by Canvas (along with many other LMSs, for example LTI can be used together with edX as either a consumer or provider [14]). The laboratory framework, is designed to suit the needs of a typical classroom (in this case approximately 30 students), thus its scalability is limited.

Testing the scalability of the designed system regarding the number of users is outside of the scope of this thesis project. However, a system might be scaled up by using larger virtual instances (vertical scaling) or by creating multiple instances (horizontal scaling). Additionally, scaling up and down of services in clouds has been investigated by others [15].

## 1.6 Structure of the thesis

Chapter 2 explains what an LMS is, introduces the LTI specification for integrating external learning applications into such systems, and presents an example of an external learning tool which is integrated with Canvas LMS. Furthermore, it presents the related technologies that were used to implement the software artifact of this project, along with projects that addressed problems related to the e-learning process in other fields of Computer Science. Chapter 3 explains the methods used to evaluate the proposed artifact. Chapter 4 presents the software artifact that was designed to facilitate student understanding of Internetworking via e-learning, and finally, Chapter 5 presents the results and the future work required to prepare the software artifact for use in production with Canvas LMS.

# Chapter 2

# Background

This chapter explains what an LMS is and how learning applications are integrated in such systems to support rich e-learning. Moreover, it introduces research artifacts that offer on-line training environments for various courses in the Computer Science domain. Lastly, it introduces those technologies that were used to design the framework that supports training events for an Internetworking course.

## 2.1 LMS

LMSs are software applications that automate the training, teaching, and administrative tasks of the learning process [2]. They have been widely adopted by higher education institutions to automate their organizational functions and provide a rich e-learning experience for both instructors and students.

Such systems are designed to provide self-guided services; rapid delivery and composition of learning material; tracking and reporting of progress through training programs, classroom, or on-line events; personalized content; and centralization and automation of administration [16]. From a learner's perspective the most common use cases of a LMS are planning ones own learning experience and collaboration with colleagues; while from an instructor's perspective the most common use cases are the design and delivery of educational content along with tracking and analysis of students' learning evolution [17].

The main functionality of a LMS concerns content organization and delivery, communication and collaboration, and assessment* of student's learning process. Some of the most commonly used features of an LMS for e-learning are video streaming of lectures, on-line notes and presentations, quizzes and practice

---

*According to Wynne Harlen and Mary James [18], formative assessment is performed by teachers during the learning process, to modify and improve the teaching and learning activities. It is based on observation of students' individual efforts and development; thus, having a qualitative and diagnostic nature. Summative assessment, performed by both instructors and students, is based on public criteria that aim to measure student's achieving of the course learning outcomes.

environments, automatic evaluation of assignments (usually exercises with predefined input and output), wikis, and discussion forums [19]. These services are either offered directly by the LMS or by integrating external applications that are designed according to specific interoperability standards. Section 2.2 describes this interoperability and integration in detail.

Although LMSs provide built-in learning applications for designing e-learning courses, their functionality is often very limited and might not suit the needs of every course. Moreover, not all LMSs support the same learning tools, nor provide the same functionality for e-learning. Fortunately, external learning tools can be integrated with multiple different LMSs, allow re-use of existing materials thus minimizing the effort for designing an e-learning course. Usually such tools are web services* that are discoverable by an LMS via the service's Uniform Resource Locator (URL) and authorization parameters (such as secret keys). The communication between the LMS and the tool is performed by exchanging messages whose format and content is defined by the interoperability specification. Section 2.3 shows several web frameworks that can be used to design external learning tools as web services.

There are several LMSs in the market (Blackboard, Moodle, Kanu, . . . ) that are used by multiple institutions. In the scope of this project the chosen learning management platform is Canvas [6]. This LMS was chosen because the system is open source, supports a well defined interoperability specification, and was selected in 2016 by KTH as their LMS.

## 2.2 LTI

Interoperability is the ability to communicate, execute programs, or transfer data among functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units [20]. An e-learning platform usually consists of several services such as course and user administration modules, and learning applications that exchange information in a formal and standardized way.

The IMS Global Learning Consortium Tools Interoperability (LTI) specification establishes a way of integrating rich learning applications (often remotely hosted and provided through third-party services) with platforms, such as LMSs, portals, learning object repositories, or other educational environments [21]. The main goal of LTI is to standardize the process of building links for sharing information and exposing functionality between external learning tools and the LMS [22]. There are two major pieces of software involved in LTI. The first is called a Tool Consumer (TC) and it refers to the software (such as an LMS) that consumes the output of

---

*In service oriented architectures, a web service is a piece of software that makes itself available over the Internet and allows third-party software to communicate with them by exchanging strictly defined messages formatted in Extensible Markup Language (XML), JavaScript Object Notation (JSON), etc.

external tools, and the second, is a Tool Provider (TP) which provides an external tool for use by the TC.

An example of a basic learning tool, is a service that accepts a request to perform a course assignment such as multiple choice question via a web form, evaluates the user's input, and returns a pass/fail grade. In this scenario, the service is the TP and Canvas LMS is the TC. A user of Canvas with administrative access (e.g., teacher), configures the integration of the external tool, a course assignment for which the tool will be launched, and finally, chooses whether the interface of the tool will be embedded in Canvas, or run in a new browser window. Figure 2.1 shows a basic flow for launching a TP from the TC. The user requests from the LMS that they want to do an assignment. This specific assignment has been configured to launch a specific LTI capable external tool together with arguments that are passed to the TP. The TP authenticates and accepts the LTI Launch request by the TC and starts a session for that particular user that allows this user to interact with the assignment.
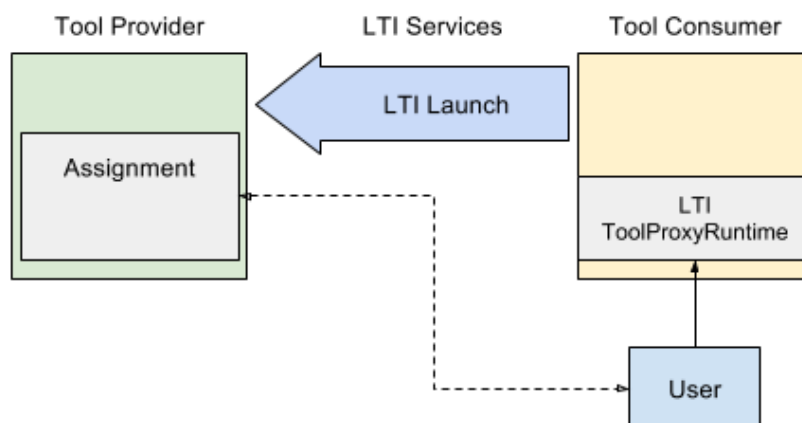


Figure 2.1: User launching an external tool

A TP often requires access to course related information, such as people, groups, memberships, courses, and outcomes. This information along with standardized ways of retrieving it are defined by the IMS Global Learning Consortium Learning Information Services (LIS) specification [23]. These services can be provided either by the TC or by a third party system. Canvas LMS implements the LTI version 1.1 which includes a subset of the LIS specification, called the LTI Basic Outcomes Service. In the example mentioned above, the information that Canvas provides to the TP when performing an LTI Launch are: how to access the LIS services, the resource identifier (assignment) for which a grade will be reported, and user information such as the unique identifier of the student. Figure 2.2 shows how a TP can communicate with LIS services to get user data and report the grade of the assignment back to the TC.
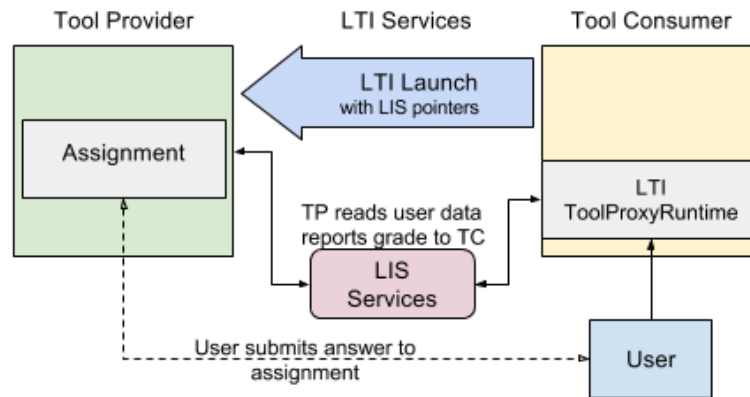
Figure 2.2: A TP using LIS services

## 2.3 Sinatra DSL

A simple web server is a piece of software designed to process Hypertext Transfer Protocol (HTTP) requests. Many web frameworks have been developed in several programming languages that allow to rapid development of web servers and applications. Amongst these, Sinatra [24, 25], is a Domain Specific Language (DSL) for writing web applications in Ruby. A Sinatra web application is organized around routes which are HTTP methods paired with a URL-matching pattern. Listing 2.1 presents a minimal sinatra application. The route "/" is paired with a `get` HTTP method. Every time this route is invoked, it provides a "Hello World!" text response.

Listing 2.1: Sinatra basic route

```ruby
# hello_world.rb
require 'sinatra'

get '/' do
  'Hello world!'
end
```

A file named `hello_world.rb` contains the code shown in Listing 2.1, which is called a route block. A route block starts with a keyword such as `get, post, put, ...` and corresponds to an HTTP method, and finishes with the keyword `end`. Executing the web application is as simple as running the command `ruby hello_world.rb`. This will start a sinatra web server on the default host (`localhost`) that listens for Transmission Control Protocol (TCP) connections on the default port (`4567`). By visiting the URL `http://localhost:4567/` with a browser, the route "/" is invoked and the response returned to the user.

A route can also utilize HTTP GET query parameters as shown in Listing 2.2. In this case, if a `course_id` is provided as a parameter of query string, then its value is loaded into the local variable `courseID`. The same concept could be applied if the route was an HTTP POST method and course_id was one of the post's parameters.

Listing 2.2: Sinatra route with HTTP GET parameters

```
get '/assignments' do
  # matches "GET /assignments?course_id=IKXXX"
  courseID = params['course_id']
  # uses course_id variable; query is optional to the / route
end
```

Sinatra also supports the use of wildcards to match all parameters of the query string. Such parameters are called splat, are symbolized with a "*" router pattern, and are accessible via the `params['splat']` array. In the Listing 2.3, the route `'/department/*/course/*'` represents the course catalog of a university. The splat parameters match the department (`informatics`) and course (`ID001`) identifiers respectively.

Listing 2.3: Wildcard route pattern

```
get '/department/*/course/*' do
  # matches /department/informatics/course/ID001
  params['splat'] # => ["informatics", "ID001"]
end
```

Templates are a text injection mechanism, that allows static text to be enriched using dynamic content (e.g., an Hyper Text Markup Language (HTML) template might contain some static text and variables, where the variables are replaced during runtime). In Sinatra a template by default is stored under the directory `./views`, and can be used in many different ways, including rendering HTML pages, constructing a JSON object as a response to an HTTP request, etc. Listing 2.4 shows the route `get '/assignments'` which stores the value of the `course_id` parameter into an instance variable `@courseID` which makes the value of this variable available for use in the template shown in Listing 2.5.

Listing 2.4: Sinatra route with template

```
get '/assignments' do
  @courseID = params['course_id']
  erb :index
end
```

Calling the `assignments` route by visiting the url `http://localhost:4567/assignments?course_id=IK1552` will parse the query parameter, invoke the `index.erb` template* stored under the directory `./views`, and substitute for the text `<%= @courseID%>` with the value of the variable

---

*Embedded RuBy (ERB) is part of the Ruby standard library, and serves as the mechanism for variable substitution within template files.

`@courseID`. The response that will be rendered by the browser will be an HTML page that contains the text "List of assignments for IK1552" in its body.

Listing 2.5: index.erb

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Assignments</title>
  </head>
  <body>
    <p>List of assignments for <%= @courseID%></p>
  </body>
</html>
```

A Sinatra route can be used to serve static files. By default, static files are served from the `./public` directory that is located under in the same directory as the application. A Sinatra application, although it is minimalistic, it is not limited to default options, thus one can configure different port numbers, root directories, custom template engines and locations, etc. Other web servers similar to Sinatra are: Flask in Python, and Netty in Java.

A collection of URL routes such as `/department/*/course/*` and `/assignments` describe a server-side web application programming interface (API), that is based on an HTTP request-response message exchange. In the context of web application development, such routes are named API endpoints and they describe the method for accessing application resources. An endpoint is consumed by a client-side application or a web service, and are either publicly accessible or protected by some sort of authorization scheme.

## 2.4 LTI tool provider

This section presents a TP written in Ruby Sinatra that implements the Basic Outcomes Service of the LTI specification. This TP is integrated into the Canvas LMS which will act as a TC. The TP has three routes (listed in Table 2.1).

Table 2.1: Routes of the TP

| | |
|---|---|
| `launch` | route for launching the external tool |
| `assignment` | route for starting an assignment |
| `report` | route for reporting the result of the assignment to Canvas LMS |

The `launch` route implements the LTI Launch functionality of the LTI specification, accepts requests for launching the external tool, and initiates a unique session per request. The `assignment` route checks for a valid session, and then returns an HTTP response with an HTML form. The form is the assignment and in this example contains a simple arithmetic question that the student has to

reply to by submitting her answer in the form's input. Finally, the `report` route validates the student's input, and reports a pass/fail grade to the TC.

This example assumes that a Canvas instructor has created an assignment and configured it to launch the TP. The following code snippets present the code implementation of the TP (inspired by lti_example from the github repository of Instructure Inc. at [26], the functionality of each route, and the XML messages that are used to communicate with the TC.

Listing 2.6 shows the code dependencies to implement the TP. First it requires the `sinatra` gem* and the `oauth` gem (used to implement the service provider, according to the LTI specification for authorization between a a TP and a TC). The `$oauth_key†` and `$oauth_secret` variables define the key and secret that is used by the TP to identify the TC. These variables are configured in a Canvas LMS when specifying the external tool. Finally the `disable :protection` statement allows for the HTML content produced by the Sinatra application to be embedded into an HTML frame of the TC, and the `enable :sessions` statement allows for session information to be used between subsequent HTTP requests to Sinatra routes.

Listing 2.6: Code dependencies and some global variables of the TP

```ruby
# dependencies
require 'sinatra'
require 'oauth'
require 'oauth/request_proxy/rack_request'

# key and secret for authenticating requests from the TC
$oauth_key = "test"
$oauth_secret = "secret"

# disable x-frame to allow embedding the TP in the TC
disable :protection

# ennable sessions for uniquely identifying students
enable :sessions
```

The `launch` route shown in Listing 2.7 is responsible for authorizing a request from the TC to launch the assignment. First it verifies the `request` against the `secret` variable. If the authorization fails, then a text message is returned to inform the Canvas user that the integration of the tool was not successful. After the authorization succeeds, the HTTP request parameters `lis_outcome_service_url` and `lis_result_sourcedid` (these correspond to the LTI LIS services) are read. The first corresponds to the TC URL that is used to report a grade for an assignment, while the latter is a unique identifier that is used to map an assignment grade to a particular student. If these parameters were not provided when Canvas invoked this route, then the request will fail. By default

---

*Ruby gems are versioned packages of ruby source code. In practice they are libraries that are hosted in public servers that make them available for download via ruby package management systems.

Canvas sets these parameters when a tool provider is correctly configured as a graded assignment. After the successful verification of the afore mentioned parameters, their values are stored in the corresponding session objects and the route redirects to the `get /assignment` route.

Listing 2.7: Launch route

```ruby
post "/launch" do
  # verify the request of the TC
  begin
    signature = OAuth::Signature.build(request, :
   consumer_secret => $oauth_secret)
    signature.verify() or raise OAuth::Unauthorized
  rescue OAuth::Signature::UnknownSignatureMethod,
        OAuth::Unauthorized
    return %{Unauthorized attempt. Make sure you used the
   consumer secret "#{$oauth_secret}"}
  end

  # Verify that this is a valid request
  # to perform an assignment
  unless params['lis_outcome_service_url'] && params['
   lis_result_sourcedid']
    return %{It looks like this LTI tool was not launched as
   an assignment, or you are trying to do the assigment as a
   teacher rather than as a a student.}
  end

  # store the relevant parameters from the launch into the
  # user's session, for access during subsequent HTTP requests
   .
  %w(lis_outcome_service_url lis_result_sourcedid).each { |v|
   session[v] = params[v] }

  # Go to the assignment
  redirect to("/assignment")
end
```

The `/assignment` route, presented in Listing 2.8, starts by validating the session variable `lis_result_sourceid`. If this parameter was not set, then the tool was not launched via the TC, hence an error text message is returned. This error message will be visible in the user's browser (either as a frame within the Canvas LMS or as a new tab on the user's browser). If the session is valid, then the route replies with an HTML form that is rendered by the user's browser. This form includes a simple arithmetic addition question and an input field for the student to reply. The form action sends the form to the `report` route using the HTTP `post` method. When the student presses the submit button within the browser, the `report` route is invoked. Note that in this listing the form has been included directly in the route block, but it could have been placed in a ruby template, such as was done for the

template in Listing 2.5.

Listing 2.8: Assignment route

```
get "/assignment" do
  # Verify the validity of the session
  unless session['lis_result_sourcedid']
    return %{You need to take this assignment through Canvas.}
  end

  # Render a form with the assignment question.
  <<-HTML
  <html>
    <head><title>Demo LTI Assignment</title></head>
    <body>
      <form action="/report" method="post">
        <p>What is the sum of 100 + 200 ?</p>
        <input name='sum' type='text' width='5' id='sum'
  required />
        <input type='submit' value='Submit' />
      </form>
    </body>
  </html>
  HTML
end
```

The `report` route, is displayed in Listing 2.9, is invoked when the student submits the form. If the form parameter `sum` is not provided, then the user is redirected (again) to the assignment via the corresponding route. Upon successful validation of the form input, an XML response message is defined and sent to Canvas via the appropriate LIS services to report the student's grade for this assignment. The format of the XML message is based upon the `imsx_POXEnvelopeRequest` class defined in the XML schema of the IMS General Web Services documentation [27] and described in the LTI 1.0 implementation guide [28].

The body of the message contains the field `sourceID` that is assigned the value of the session variable `#session['lis_result_sourcedid']` , while the resultScore field that corresponds to the assignment's grade and has the value 1 in the `textString` subfield if the provided sum was 300 or 0 otherwise. The corresponding assignment was configured earlier in Canvas to accept a maximum of 1 point for the grade for this assignment.

The message is signed according to the OAuth 1.0 protocol* using the same consumer key and secret that were provided during the LTI launch request (`launch` route). The message is posted synchronously to the Canvas LIS service

---

*OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end-user). OAuth also provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (typically, a username and password pair) by using user-agent redirections [29].

defined by `session['lis_outcome_service_url']` using a Multipurpose Internet Mail Extensions (MIME)[†] encoding, and the response is stored in the `response` variable. Because the post was done synchronously, the code will wait until the response to this post is received. Thus the body of the response can be used to compute the message to be displayed to the user via their browser.

Listing 2.9: Report the assignment grade to Canvas

```
post "/report" do
  sum = params['sum']
  if !sum || sum.empty?
    redirect to("/assignment")
  end

  # now post the score to canvas. Make sure to sign the POST
    correctly with
  # OAuth 1.0, including the digest of the XML body. Also make
    sure to set the
  # content-type to application/xml.
  xml = %{
<?xml version = "1.0" encoding = "UTF-8"?>
<imsx_POXEnvelopeRequest xmlns = "http://www.imsglobal.org/lis
  /oms1p0/pox">
  <imsx_POXHeader>
    <imsx_POXRequestHeaderInfo>
      <imsx_version>V1.0</imsx_version>
      <imsx_messageIdentifier>12341234</imsx_messageIdentifier
 >
    </imsx_POXRequestHeaderInfo>
  </imsx_POXHeader>
  <imsx_POXBody>
    <replaceResultRequest>
      <resultRecord>
        <sourcedGUID>
          <sourcedId>#{session['lis_result_sourcedid']}</
  sourcedId>
        </sourcedGUID>
        <result>
          <resultScore>
            <language>en</language>
            <textString>#{sum == 300 ? 1 : 0}</textString>
          </resultScore>
        </result>
      </resultRecord>
    </replaceResultRequest>
  </imsx_POXBody>
</imsx_POXEnvelopeRequest>
```

---

[†]The MIME-type is a two-part identifier for file formats and format of contents transmitted via the Internet.

```
  }
  consumer = OAuth::Consumer.new($oauth_key, $oauth_secret)
  token = OAuth::AccessToken.new(consumer)
  response = token.post(session['lis_outcome_service_url'],
   xml, 'Content-Type' => 'application/xml')

  headers 'Content-Type' => 'text'
  %{
Your score has #{response.body.match(/\bsuccess\b/) ? "been
   posted" : "failed in posting"} to Canvas. The response was:
#{response.body}
  }
end
```

Lastly the contents of `reponse` are evaluated and checked to a certain degree
whether posting the grade was successful or not, and then a text message is sent
to the user to be rendered by her browser informing her about the status of
posting the grade to Canvas. The response of a successful post is highlighted in
Listing 2.10 in the `imsx_codeMajor` xml field.

Listing 2.10: XML response from Canvas

```
<?xml version="1.0" encoding="UTF-8"?>
<imsx_POXEnvelopeResponse xmlns="http://www.imsglobal.org/
   services/ltiv1p1/xsd/imsoms_v1p0">
  <imsx_POXHeader>
    <imsx_POXResponseHeaderInfo>
      <imsx_version>V1.0</imsx_version>
      <imsx_messageIdentifier/>
      <imsx_statusInfo>
        <imsx_codeMajor>success</imsx_codeMajor>
        <imsx_severity>status</imsx_severity>
        <imsx_description/>
        <imsx_messageRefIdentifier>12341234</
  imsx_messageRefIdentifier>
        <imsx_operationRefIdentifier>replaceResult</
  imsx_operationRefIdentifier>
      </imsx_statusInfo>
    </imsx_POXResponseHeaderInfo>
  </imsx_POXHeader>
  <imsx_POXBody><replaceResultResponse/></imsx_POXBody>
</imsx_POXEnvelopeResponse>
```

## 2.4.1 Integration of an external application into Canvas LMS

The text above presented how to develop a simple LTI provider that supports
graded assignments. The Canvas LMS Graphical User Interface (GUI) allows the
integration of external applications via different options, such as manual
configuration forms, launch URLs, and pasting in XML entries. This section

presents how to configure an external tool using a manual configuration form via the `Settings->Apps->External Apps->Add App` menu for a Canvas course. Here we assume that an instructor wishes to add an external app for a particular course. The input form shown in Figure 2.3 is loaded. The instructor inputs a name for the application, the LTI Launch URL, and the consumer key and secret.



Figure 2.3: Adding an external application to Canvas

After adding this external tool, the instructor creates a new assignment, configures it to launch the application within Canvas, or using an external window (as shown in Figure 2.4), and then specifies a grading scheme. Once the assignment is configured and published in Canvas, a student can complete this assignment via the course page. Section 2.5 explains how to integrate external applications using URLs and XML configuration.



Figure 2.4: Configuring an assignment to use an external tool

### 2.4.2 Securing the connection between a TP and a TC

The communication between the Canvas LMS and external application tools is by
default expected to be performed using the Hypertext Transfer Protocol
Secure (HTTPS)* protocol. In the example presented in previous section, the
communication between the TP and the TC was over HTTP, hence Canvas
generated a corresponding error while launching the TP. The Sinatra web-server
can be easily configured to listen for HTTPS connections on a specific port.
Listing 2.11 shows such a configuration of the Sinatra web server (named
Webrick). HTTPS requires a TLS certificate which for the purposes of this
example was issued and signed using the OpenSSL [30] cryptography and TLS
toolkit, rather than a trusted third party Certificate Authority (CA).

Listing 2.11: TLS configuration of a Sinatra application

```ruby
require 'sinatra/base'
require 'webrick'
require 'webrick/https'
require 'openssl'

CERT_PATH = '/opt/CA/'

webrick_options = {
  :Port               => 8443,
  :Logger             => WEBrick::Log::new($stderr, WEBrick::
   Log::DEBUG),
  :DocumentRoot       => "/ruby/htdocs",
  :SSLEnable          => true,
  :SSLVerifyClient    => OpenSSL::SSL::VERIFY_NONE,
  :SSLCertificate     => OpenSSL::X509::Certificate.new(File.
   open(File.join(CERT_PATH, "cert.pem")).read),
  :SSLPrivateKey      => OpenSSL::PKey::RSA.new(File.open(File
   .join(CERT_PATH, "key.pem")).read),
  :SSLCertName        => [ [ "CN", '127.0.0.1' ] ]
}

class MyServer  < Sinatra::Base
    post '/' do
      "Hellow, world!"
    end
end

Rack::Handler::WEBrick.run MyServer, webrick_options
```

---

*HTTPS is a protocol for communication over HTTP within a connection encrypted by
Transport Layer Security (TLS). TLS uses a public and a private encryption key to generate
a session key which is used to encrypt the data flow between client and server. An HTTP message
is encrypted prior to transmission and decrypted upon arrival.

Listing 2.12 shows how to generate a TLS certificate using the OpenSSL command line tool. The command is `openssl req` and it takes several arguments such as `-new` (request new certificate), `-x509` (format of the public key), `-extensions v3_ca` (the extensions to add for a self signed certificate, shown in the corresponding block of Listing 2.13, `-keyout key.pem` (the output file for storing the key), `-out cert.pem` (the output file for storing the self-signed certificate), `-days 365` (the number of days until the certificate expires), and finally the sample configuration file `openssl.conf` for reading the default values.

Listing 2.12: Generating a self signed TLS certificate and encryption key

```
openssl req -new -x509 -extensions v3_ca -keyout key.pem -out
    cert.pem -days 365 -config ./openssl.conf
```

The OpenSSL configuration shown in Listing 2.13, is a sample file containing default values for generating a TLS certificate and a public key file, and is available for download in Markus Redivo's page "Creating and Using SSL Certificates" [31]. More details regarding the use of the `req` command of the OpenSSL toolkit can be found in the corresponding man page [32], and information about the configuration file can be found in Phil Dibowitz's blog page "Openssl.conf walkthru" [33].

Listing 2.13: Sample OpenSSL configuration for issuing SSL/TLS certificates

```
---Begin---
# OpenSSL configuration file.

# Establish working directory.
dir = .

[ ca ]
default_ca    = CA_default

[ CA_default ]
serial        = $dir/serial
database      = $dir/index.txt
new_certs_dir = $dir/newcerts
certificate   = $dir/cacert.pem
private_key   = $dir/private/cakey.pem
default_days  = 365
default_md    = md5
preserve      = no
email_in_dn   = no
nameopt       = default_ca
certopt       = default_ca
policy        = policy_match

[ policy_match ]
countryName            = match
stateOrProvinceName    = match
organizationName       = match
```

```
organizationalUnitName = optional
commonName             = supplied
emailAddress           = optional

[ req ]
default_bits        = 1024      # Size of keys
default_keyfile     = key.pem   # name of generated keys
default_md          = md5       # message digest algorithm
string_mask         = nombstr   # permitted characters
distinguished_name = req_distinguished_name
req_extensions      = v3_req

[ req_distinguished_name ]
# Variable name          Prompt string
#--------------------   ----------------------------------
0.organizationName      = Organization Name (company)
organizationalUnitName  = Organizational Unit Name (department, division)
emailAddress            = Email Address
emailAddress_max        = 40
localityName            = Locality Name (city, district)
stateOrProvinceName     = State or Province Name (full name)
countryName             = Country Name (2 letter code)
countryName_min         = 2
countryName_max         = 2
commonName              = Common Name (hostname, IP, or your name)
commonName_max          = 64

# Default values for the above, for consistency and less typing.
# Variable name                Value
#--------------------         ------------------------------
0.organizationName_default  = The Sample Company
localityName_default        = Metropolis
stateOrProvinceName_default = New York
countryName_default         = US

[ v3_ca ]
basicConstraints        = CA:TRUE
subjectKeyIdentifier    = hash
authorityKeyIdentifier = keyid:always,issuer:always

[ v3_req ]
basicConstraints        = CA:FALSE
subjectKeyIdentifier = hash


----End----
```

## 2.5 LTI applications

Edu App Center [34] is an open database for learning tools maintained by Instructure [35] and among its several services, it offers a collection of open learning applications that implement the LTI specification. These applications can be integrated with different LMSs. The user can apply filters to locate an appropriate tool and can browse tutorials about integrating a tool with the LMS of their choice. Often these tools are hosted by third party services (e.g GitHub, Youtube, Turnitin). The goal of Edu App Center is to enable instructors to easily configure these external applications to their courses, thus providing and fostering a market place for LTI applications.

Section 2.4.1 presented how an instructor can integrate a Ruby Sinatra external application into Canvas LMS using a web form. This approach is limited to the functionality of Canvas LMS. An alternative method for integrating external applications via XML configuration can be used across different LMSs. Edu App Center offers such configurations for every LTI tool listed in the marketplace. Additionally, it provides the XML Config Builder service, that allows instructors to generate XML for integrating custom built external LTI applications into different LMSs. Listing 2.14 shows an example of such XML entry (generated by the Edu App Center's XML Config Builder) that was used to integrate the Ruby Sinatra application (presented in the previous section) into Canvas.

First, the XML version and the charset encoding are defined. Then the `cartridge_basiclti_link xmlns` specifies that this is an LTI link that can be used for integrating an external application. This block contains the whole XML configuration. It starts by defining the IMS Global XML schema that is used to describe this entity. Then the LTI Launch URL is specified (`blti:launch_url`), and it is followed by metadata, regarding the title (`blti:title`) and description (`blti:description`) of the external application. Finally, it defines a block for LTI extensions (`blti:extensions platform`) that specifies the LMS platform to act as a TC for this TP. This block of XML code can contain information that is specific to each LMS that is supported by the TP.

Listing 2.14: XML configuration of an external application for Canvas

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cartridge_basiclti_link xmlns="http://www.imsglobal.org/xsd/
   imslticc_v1p0"
  <!-- Definition of the XML Schema -->
  xmlns:blti = "http://www.imsglobal.org/xsd/imsbasiclti_v1p0"
  xmlns:lticm ="http://www.imsglobal.org/xsd/imslticm_v1p0"
  xmlns:lticp ="http://www.imsglobal.org/xsd/imslticp_v1p0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.imsglobal.org/xsd/
   imslticc_v1p0 http://www.imsglobal.org/xsd/lti/ltiv1p0/
   imslticc_v1p0.xsd
  http://www.imsglobal.org/xsd/imsbasiclti_v1p0 http://www.
   imsglobal.org/xsd/lti/ltiv1p0/imsbasiclti_v1p0.xsd
  http://www.imsglobal.org/xsd/imslticm_v1p0 http://www.
   imsglobal.org/xsd/lti/ltiv1p0/imslticm_v1p0.xsd
  http://www.imsglobal.org/xsd/imslticp_v1p0 http://www.
   imsglobal.org/xsd/lti/ltiv1p0/imslticp_v1p0.xsd">

  <!-- The LTI Launch url -->
  <blti:launch_url>http://192.168.39.39:4567/launch</
   blti:launch_url>
  <!-- Title of the External Application -->
  <blti:title>Arithmetic Assignment</blti:title>
  <!-- Description for the external application -->
  <blti:description>Sample arithmetic assignment tool</
   blti:description>
  <-- Configuration specific to the TC -->
  <blti:extensions platform="canvas.instructure.com">
    <lticm:property name="privacy_level">public</
   lticm:property>
  </blti:extensions>
</cartridge_basiclti_link>
```

## 2.6 Previous efforts to provide on-line exercise material

Traditional practice events in Computer Science involve laboratory environments and exercises based on physical or virtual hardware and domain specific software. One of the problems is creating and managing these environments. Previously such material was packaged in virtual machines or run in an isolated environment (such as a sandbox or linux container as will be described in Section 2.7).

## 2.7 Linux Containers

A container is a light weight operating system running inside the host system, executing instructions native to the Central Processing Unit (CPU), eliminating the need for instruction level emulation or just in time compilation [36]. Linux Containers (LXC) [37] is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a host using a single Linux kernel. Its purpose is to virtualize a single application rather than a whole operating system inside a virtual machine. LXC uses cgroups* to isolate resources (such as CPU, memory, network, etc.) and namespaces† to isolate the application from the operating system [39].

Docker [40] was initially a Linux container engine that provides the ability to manage containers as self contained images. Docker utilizes LXC for the container implementation, has image management capabilities, and implements a Union File System (UnionFS). It features resource isolation via cgroups and namespaces, network and file system isolation through LXC functionality, and allows managing the lifecycle of a container [36]. Although docker initially utilized LXC as the only execution driver for resource isolation, lately it introduced libcontainer [41], which includes its own implementation for resource isolation, but also has bindings to leverage other technologies (such as LXC, libvirt-lxc [42], and systemd-nspawn [43]), thus libcontainer realizes a cross-system abstraction layer for packaging, delivering, and running applications in isolated environments. The implementation and functionality of libcontainer is defined by the Open Container Initiative (OCI) [44] specification which defines the image formats, the image management interface, and the container runtime life-cycle.

Docker leverages a client-server architecture. The server is called a docker daemon, and it is responsible for the container's runtime environment. It also has capabilities for building, running, and distributing docker containers. The Docker client is a user interface for communicating with the docker daemon. The client has several implementations, including a command line tool [45] and the Docker Remote API [46]. The Docker ecosystem includes different technologies and tools for managing images, container and application runtime, infrastructure deployment and orchestration, etc. The Docker Hub is an image registry that stores container images in a similar way as traditional package management stores software artifacts. An image is part of a repository and has an author and a version, thus making the image and its configuration easy to distribute and discover.

---

*Control groups (cgroups) is a Linux kernel feature that is responsible for managing resources such as CPU, memory, disk I/O, network, etc.

†A namespace wraps a global system resource (process IDs, mount points, network devices, network stacks, ports, etc.) in an abstraction that makes it accessible to the processes. Within a namespace each process has its own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes [38].

Listing 2.15 illustrates how a container image can be downloaded from the Docker Hub using the command line interface of the docker daemon. The command `docker pull ubuntu:14.04` requests a download of the image of Ubuntu from the repository that is tagged with version 14.04. To realize this pull, the Docker daemon connects to the Hub and then requests this particular image of that repository, and starts downloading the image together with its configuration and dependencies. Finally, after the downloading is complete, the Docker daemon creates a hash string of the image using the Secure Hash Algorithm (SHA) algorithm. Subsequently this hash is used uniquely identify the image in the local registry of this docker daemon.

Listing 2.15: Docker pull command

```
$: docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu

ba76e97bb96c: Pull complete
4d6181e6b423: Pull complete
4854897be9ac: Pull complete
4458f3097eef: Pull complete
9989a8de1a9e: Pull complete
Digest: sha256:062bba17f92e749bd3092e7569aa0\
    6c6773ade7df603958026f2f5397431754c
Status: Downloaded newer image for ubuntu:14.04
```

Using the command line client, docker can list all downloaded images along with a set of metadata for these images. Listing 2.16 shows the output of the command `docker images`, which contains the name of the repository, the repository tag, a unique identifier of the image, and additional information (such as when the image was created and stored in the Docker Hub), and its size.

Listing 2.16: Docker images command

```
$: docker images
REPOSITORY    TAG        IMAGE ID       CREATED       SIZE
ubuntu        14.04      4d44acee901c   3 days ago    187.9 MB
```

The container runtime, defines the different states of a container: created, started, paused, stopped, and deleted. In order to run an application inside an isolated environment, first a container has to be created from an existing image and then started. Listing 2.17 shows the command `docker run` which specifies the execution of a container from a particular image and causes it to execute a particular application (in this case `/bin/bash`).

Listing 2.17: Docker run command

```
$: docker run -t -i ubuntu:14.04 /bin/bash
```

In more detail, the command causes the runtime to create a container from the image `ubuntu:14.04`, and configures it according to the specified arguments. The

25

command argument `-t` requires allocates a pseudoterminal (pty) [47], and the argument `-i` attaches the standard input and output to this pseudoterminal. Finally, the container starts and executes the command `/bin/bash`.

Listing 2.18 illustrates the `docker ps` command which lists the containers that are in the running state. The output of the command includes information such as the unique identifier of the container, the container image, the command that is running, and other information such as when the container was created it, when it started running, what port bindings the container has with the host operating system, and a unique name.

Listing 2.18: Docker ps command

```
$: docker ps
CONTAINER ID   IMAGE          COMMAND        CREATED        STATUS        PORTS   NAMES
91af84830636   ubuntu:14.04   "/bin/bash"    3 seconds ago  Up 2 seconds
    lonely_lichterman
```

The commands presented previously are just a subset of those available via the command line interface of the docker client. The complete set of commands can be found by running docker without any arguments or with the argument "help". Figure 2.5, from the documentation about the Docker Remote API, shows a state diagram of a container, along with the various commands and events that are responsible for containers transitioning between different states.



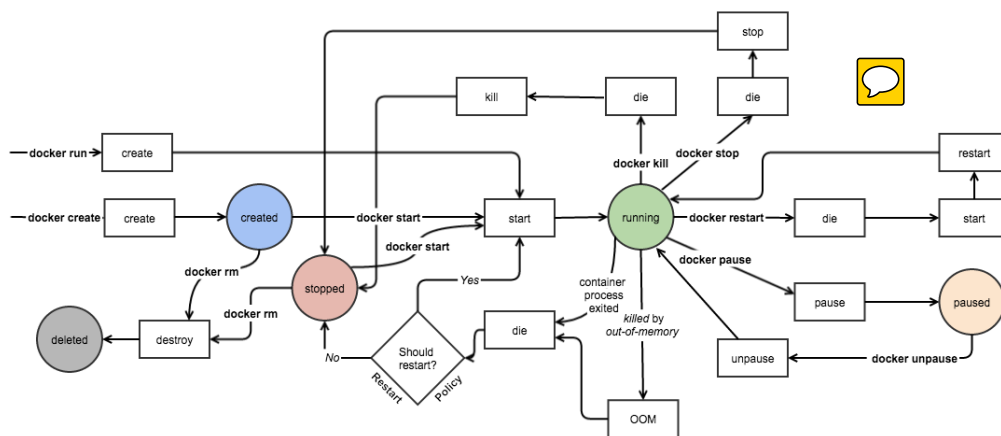Figure 2.5: States of the container lifecycle

Listing 2.17 showed how to run the bash shell process inside a linux container. The code snippets of Listings 2.19 and 2.20 illustrate how one can install a package in the operating system of the container and then create a new image of the resulting container (outside of the container).

Listing 2.19: Installing a package in the container Operating System

```
root@91af84830636:/# apt-get install traceroute
```

Listing 2.19 shows the user `root` executing the `apt-get` command in a `bash` terminal of a running container with identifier **91af84830636**. Using the apt package manager of Ubuntu, the root user installs the traceroute package. Later this running container is used to create a new image, that will contain the current state of this container (i.e., the container that now has traceroute installed in it).

Listing 2.20: Create a new docker image out of a running container

```
$: docker commit -m "traceroute-package" -a "KTH" 91af84830636
    my-ubuntus:traceroute
```

The command `docker commit` accepts a `-m` parameter containing a commit message, a `-a` parameter specifying the author of this commit (in this case "KTH"), the id of the container that will be used to create a new image (in this case 91af84830636), the name of the repository (my-ubuntu), and the reference tag for this repository (:traceroute). Executing the command `docker images` as shown in Listing 2.21, will verify that the image was created.

Listing 2.21: List the docker images, shows the newly created image

```
$: docker images
REPOSITORY    TAG           IMAGE ID         CREATED          SIZE
ubuntu        14.04         4d44acee901c     3 days ago       187.9 MB
my-ubuntus    traceroute    1261c79eb3da     4 seconds ago    166.9 MB
```
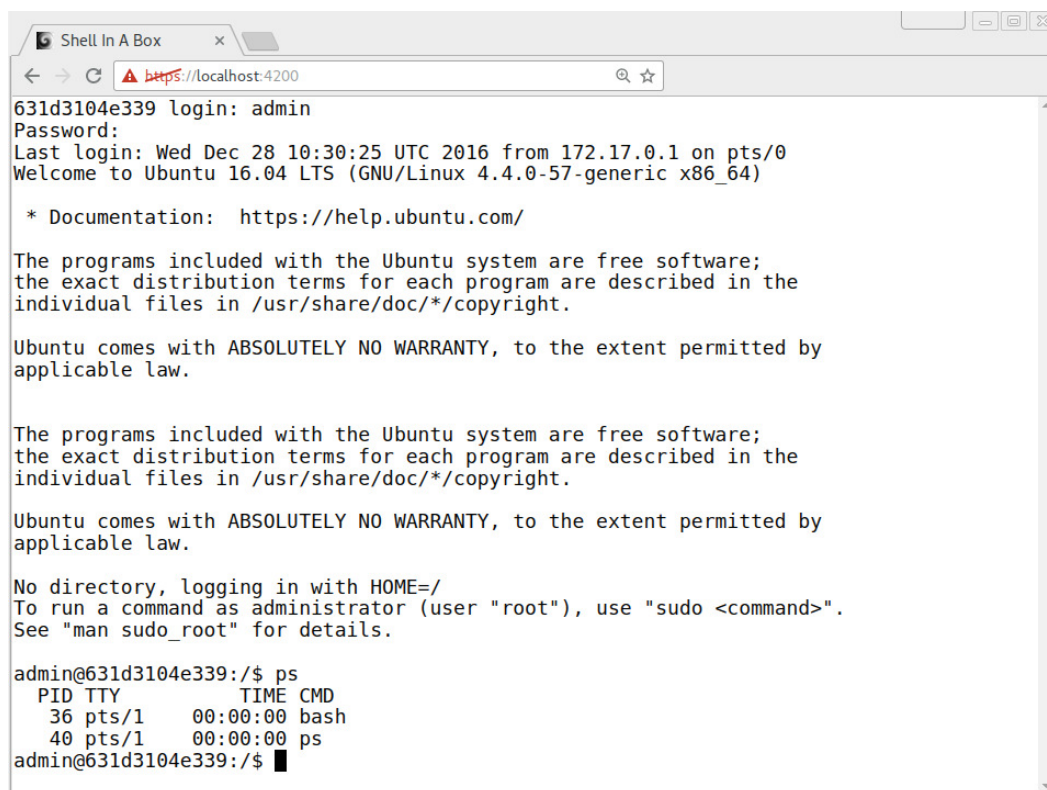
Linux containers can be used to create pre-configured machines for laboratory assignments of an Internetworking course. By creating container images tailored to the needs of each assignment, a student can focus on the exercise, while avoiding details that are not relevant to the learning process. A software solution that supports creating images and running containers on demand, can be very useful for e-learning, as it takes a student just a few seconds to access a unique laboratory environment via her web browser.

## 2.8 Web based shell emulators

When it comes to e-learning assisted by LMSs, students are used to performing most of their learning tasks via their web browser. Using pre-configured laboratory environments based on docker images entails the same risks as traditional labs, as the student has to install docker and manually execute a series of commands before she will be able to focus on the learning process. An alternative solution would be to support such environments in a remote server, and then simply provide the student access to the remote environment via a web browser. The software that provides access to a linux shell via a web browser is often called a web based terminal emulator. The technology that provides communication between the server (the terminal emulator) and the client (the web browser) is called Web-based Secure Shell (SSH). The server side of the implementation involves a web application that

accepts requests for keyboard events and forwards these keyboard events to a secure shell client communicating with the connected SSH server. The terminal output is either passed to the client where it is converted into HTML via JavaScript or it is translated into HTML by the server before it is transmitted to the client [48].

There are several implementations of web based shell emulators, such as GateOne [49] and Shell In A Box [50]. The latter, implements a web server that can export arbitrary command line tools to a web based terminal emulator. This emulator is accessible to any JavaScript and Cascading Style Sheets (CSS) enabled web browser. The server listens on a specified port and publishes services that are displayed by a VT100 [51] emulator implemented as an Asynchronous JavaScript and XML (AJAX) [52] web application. Figure 2.6 shows the web based emulator running in a web browser that enables the user to access the remote system via an SSH session. In this case the Shell In A Box web server is a process running on a docker container based on the `ubuntu:16.04` docker image, and is listening for secure TLS connections on port `4200`.



Figure 2.6: Shell In A Box emulator running in a web browser window

The default configuration settings of the server require a TLS certificate for the server to start. If no certificate is provided, a self signed certificate is generated. In addition to the certificate, Shell In A Box requires users that want to access the

linux server via an SSH session to authenticate themselves using a username and a password. Such credentials are also passed as parameters to the server startup process.

The docker image was configured to run the Shell In A Box web server according to the instructions of the GitHub repository *docker-shellinabox*[53] of the Github user *sspreitzer*. This repository, mentions two different methods of acquiring the docker image. The first downloads the image from the Docker image registry using the remote image repository `sspreitzer/shellinabox`*. The downloading of the image is initiated by the `docker pull` command as explained in the previous section. The second method, specifies configuration rules to use when building the image in a local image repository with the `docker build` command.

Figure 2.6 shows that the emulator is accessible via the URL `https://localhost:4200`, where `localhost` is the host system that is running the Docker daemon, and `4200` is a TCP port of the host system that is reserved by Docker and is used to forward network packets to the container that is running the Shell In A Box web server process, and is listening for connections on the container's TCP network port `4200`. When Docker is installed on a Linux host, a network interface named `docker0` is created. The `docker0` network interface is actually an Ethernet bridge* that enables packet transmission between physical and virtual network interfaces [55], and enables the host machine to receive and send packets to containers connected to this bridge interface. Additionally, the docker server has functionality that allows a network port of the host system to be bound to a network port of the container. For example, the `docker run` command accepts a parameter `-p IP:host_port:container_port` which specifies which host port should bind to a container port. The command below shows how to run a container (running a Shell In A Box web server process) from the image repository `sspreitzer/shellinabox` with version `latest`, and map the TCP port 4200 of the host system to the TCP port 4200 of the container.

```
docker run -p 4200:4200 -e SIAB_PASSWORD=123 -e SIAB_USER=
    admin -e SIAB_SUDO=true sspreitzer/shellinabox:latest
);
```

The parameter `-e` specifies environment variables that are saved in the in the linux operating system of the container during its creation. Those environment variables[†] are parsed by the Shell In A Box web server initialization script to configure the web server, the authentication credentials, and `sudo` access for the Linux user.

---

*The `sspreitzer/shellinabox` image is based on the Ubuntu 16.04 Linux operating system

*A bridge is a way to connect two Ethernet segments together in a protocol independent way. Packets are forwarded based on Ethernet address, rather than IP address (as a router would do). Since forwarding is done at Layer 2, all protocols can go transparently through a bridge [54].

†The environment variables are explained in detail in the documentation contained in the GitHub repositories referenced above.

## 2.9   Related work

The support for interoperability specifications by several LMSs has allowed rapid experimentation and implementation of external application frameworks that offer a variety of on-line training events for various Computer Science courses. This section presents some of these frameworks and describes how they are relevant to this project.

### 2.9.1   EDURange

Designing on-line training environments for the field of cyber security requires overcoming some technical constraints, such as high availability and scalability, and pedagogical limitations, such as teaching analysis skills to understand complex systems and concepts via practicing [3]. EDURange addresses these issues by designing an open source framework that provides interactive security exercises in an elastic cloud environment [56].

EDURange is a software framework, designed to work on Amazon Elastic Compute Cloud (EC2) [57]. It allows teachers to easily build and scale dynamic virtual environments to host cybersecurity training [58]. This framework provides ease of use for instructors, by offering the flexibility to specify exercises at a high level and allowing the instructor to configure different aspects of the training scenarios in order to provide a tailored learning experience that focuses on analysis skills.

### 2.9.2   GLUE!

Group Learning Uniform Environment (GLUE!) is a middle-ware integration architecture that aims to standardize the integration of existing external learning tools into several LMSs [59]. It facilitates the instantiation and enactment of collaborative learning situations within LMSs, by using the distinctive administrative features of these systems to manage users and groups. LTI and the Sharable Content Object Reference Model (SCROM) are two specifications for the integration of external learning tools into an LMS. However, each LMS usually supports only a single interoperability specification; thus, developing a universal external tool requires a substantial development effort to support the different interoperability standards. In contrast, GLUE! proposes a software architecture that takes advantage of the common integration features of LMSs to integrate multiple existing learning tools into multiple LMSs.

### 2.9.3   INGInious

Programming exercises are the most common form of practice for students learning Computer Science. Traditionally, the evaluation of these exercises, requires grading of reports, reading source code, and testing source code, thus making it time consuming, especially for large classes (i.e., large numbers of

students).  INGInious [8, 60, 61, 62] is a software framework that empowers
instructors to easily construct coding tasks and it supports automatic evaluation
and grading of the code, thus providing both students and teachers with
constructive feedback.

The framework consists of two main components: the frontend and the backend.
The frontend provides a web interface where students perform programming tasks
and an administration module that allows instructors to design these tasks. The
backend is responsible for running and grading the code inside remote isolated
Linux containers. Each container is specifically built for a particular programming
language, according to configuration provided by the instructor or the administrator
of the system, thus supporting the evaluation of tasks written in any programming
language that runs in a Linux environment.

One of the main features of INGInious is that the frontend component can be
used either as a stand-alone web application or as an external learning tool that
is integrated into an LMS using the LTI specification. Additionally, the backend
component scales horizontally very easily, since it utilizes a docker container for
every task request, therefore it is suitable for MOOC platforms.

A programming task in INGInious is designed using a configuration file
(`task.yaml`) that identifies the problem to be solved by the student, and the
evaluation process, a template file (template.py) that presents the task to the
student, and defines the input field for the code, and finally, a file (`run`) that
executes the student code, and validates the output. The following code samples
show the minimum configuration required by the instructor, to design a simple
"Hello World" task in Python. Listing 2.22 is the `task` file. It starts with
key-value pairs that are used to describe the `name` and `context` of the task.

Listing 2.22: Definition of a task in task.yaml

```
name: "Hello World!"
context: "In this task, you will have to write a python script
    that displays 'Hello World!'."
problems:
  question1:
    name: "Let's print it"
    header: "def func():"
    type: "code"
    language: "python"
limits:
  time: 10
  memory: 50
  output: 1000
environment: "default"
```

Then it defines the **problems** that have to be solved to complete this task. Each problem has a unique name within the task (**question1**) and a series of metadata such as the programming language to be used for solving the problem, and the text input to print in the input form. Finally it contains other metadata that defines the resources of the virtual environment that will be used to evaluate the code.

Listing 2.23: Code input of question1 in template.py

```
def func():
  @ @question1@@

func()
```

Listing 2.23 defined the input into field in which the student will input their code. Finally, the **run** file defined in Listing 2.24, is a shell script, that parses the input code using the INGInious commands **parsetemplate**, then evaluates the expected output against the results of the input function using the command **run_student**. Finally it prepares the result of the task using the **feedback** command.

Listing 2.24: Evaluation of student code by the run file

```
#! /bin/bash

# Parse the template and put the result in studentcode.py
parsetemplate --output studentcode.py template.py

# Verify the output of the code...
output=$(run_student python studentcode.py)
if [ "$output" = "Hello World!" ]; then
  # The student succeeded
  feedback --result success --feedback "Success!"
else
  # The student failed
  feedback --result failed --feedback "Your output is $output"
fi
```

Detailed information for specifying a task in INGInious platform can be found in the official teacher documentation [63]. As part of the research in this thesis project, the LTI component of INGInious was configured with Canvas LMS, to perform sample programming tasks such as the "Hello World!" code that was explained earlier.

## 2.10   Summary

Canvas LMS is an open source system that aims to assist in every aspect of the learning process. It offers functionality for e-learning activities such as rich media, interactive quizzes, methods for automatically evaluating assignments, and finally allows developers to design and integrate their own learning tools via the LTI specification.    The LTI specification standardizes the method of integrating external learning applications in LMSs via XML configurations, and allows the LMS to exchange structured messages with a TP to share information such as user sessions, and learning outcomes.

LTI is only one of the several specifications for integrating learning applications into LMSs. GLUE! is a middleware implementation that supports the integration of external learning tools into different LMS that implement different specifications.

Designing assignments for an Internetworking course relies heavily on laboratory environments. Creating and managing such environments can easily be performed by using Linux Containers. Docker offers a high level API that allows to create container images with provisioned software, tailored to the requirements of different assignments. The Docker runtime can nearly instantly create and execute software realizing a particular laboratory environment. Using web based shell emulators, students can access the environment and focus on the learning process, rather than configuring the environment themselves.

Similar approaches that address the problem of virtual laboratory environments, and automatic assignment evaluation have been proposed by researchers in other fields of Computer Science. Several of these approaches were evaluated, and provided useful guidelines for designing the software artifact of this project. The Edurange project focuses on devising a set of exercises that train students in the Cybersecurity domain. Moreover, if offers a method for deploying the framework in cloud infrastructures, to increase availability of the system for students and instructors, and also reduce the cost of hosting the framework for educational institutions.   The Inginious framework focuses on providing an environment for evaluating coding assignments in all programming languages whose runtime is supported by the linux kernel. The system offers high availability for evaluating code using unit tests, and the actual evaluation is performed within a docker container.

# Chapter 3

# Methodology

This thesis project is carried out using the Design Science research method. This type of research focuses on the design and construction of IT artifacts that have utility in the real world, in this case as an application environment, and aim to improve domain-specific systems and processes. In the context of this research, the real world problem is the lack of interactive virtual laboratory environments in the form of e-learning tools.

## 3.1  Research Process

Vijay Vaishnavi and Bill Kuechler in their book *Design Science Research in Information Systems* [64] describe the process for performing Design Science Research in the following five steps: Awareness of the Problem, Suggestion, Development, Evaluation, and Conclusion. In the scope of this project, the first two steps are reflected in the Introduction and Background chapters (i.e., Chapters 1 and 2). The literature study that was performed, provided understanding of the problem, of how other researchers have addressed similar problems, and how existing technologies can be combined to devise a solution for the problem addressed by this thesis project. The Development step is reflected in Chapter 4: Implementation, which describes the designed software artifact. The Evaluation step is addressed in Section 4.4, evaluates the functionality of the artifact against a set of criteria (listed in Section 3.2). Finally, the Conclusion step (covered in Chapter 5), summarizes the results, and proposes a series of actions to be taken as part of the future work of this project.

## 3.2  Evaluation Process

The literature study that was carried out within the scope of this project revealed two important software solutions (EduRange and INGinious) that address similar problems in different domains of Computer Science. Further analysis of their

functionality and implementation inspired the work of this project and lead to a number of high level requirements. These requirements are:

- The laboratory environments can be designed using Docker containers, in a similar way that INGInious uses them to perform the evaluation of student assignments. A laboratory environment can be realized by creating a container image which includes all software required for a given Internetworking assignment.

- High availability of these laboratory environments can be achieved by creating and running a docker container for each student session. This approach was utilized by both INGInious (for the evaluation of each coding assignment) and Edurange (which relies on preconfigured virtual machines that are used to facilitate Cybersecurity training).

- The instructor should be able to dynamically update the underlying software and the assignments, similarly to the way INGInious creates a new assignment.

- The system's design should not be specific to a particular cloud infrastructure provider. This is facilitated because of the fact that the Docker runtime is supported by most linux operating system distributions which can run on dedicated or virtual hardware.

Furthermore, a series of goals were selected to be used as guidelines for the system's design. These guidelines focus on the interaction of the two main user roles of the system: the instructor and the student. These guidelines are:

- The instructor should have complete control over the software used for a particular assignment (for example, install the software and create a container image that will be used for a particular assignment).

- The instructor should always know which container images exist in the system, and should have sufficient privileges to delete and create these images.

- The system should provide a way for the instructor to access a laboratory environment, in similar to the way a student is expected to access it.

- The system should provide a suitable configuration for the instructor to create an assignment in Canvas LMS and connect it with a particular container image.

- The student should be able to launch a laboratory environment from a Canvas LMS, by simply pressing the assignment button. The resulting container should be available to the user (almost) instantly (as shown in Section 4.3.2).

The evaluation of the software artifact was performed in two steps. First, the system was evaluated against the requirements mentioned earlier to validate whether the

solution is aligned with the goals of this project, and then, additional evaluation methodologies such as unit testing were used to test that the implemented code was performed as intended.

# Chapter 4

# Implementation

The artifact that was designed within the scope of this thesis work consists of two different modules: a TP and a Tool Client. The TP enables students to access a laboratory environment via LTI integrations with a Canvas LMS, while the Tool Client provides an administrative tool which exposes functionality enabling the instructor to preconfigure the laboratory environments and configure the integration with the LMS acting as a TC. Figure 4.1 presents a high level overview of these two types of users interactions with the system.
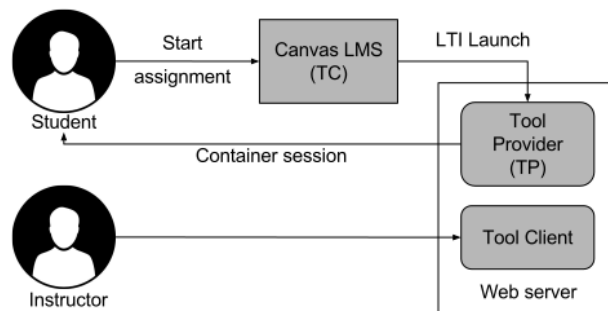


Figure 4.1: High Level Overview of the System Architecture

The TP and the Tool Client are **not** separate systems, but different components of the same web server. This co-location enables them to share common functionality such as the container runtime and management of user sessions.

## 4.1 Software architecture

Section 2.3 introduced an example of a web server which had the role of a TP that accepted and authenticated requests from a Canvas LMS to launch assignments. Similarly to that approach, an HTTP web server was used to support the

functionality of the LTI Tool Client and the LTI Tool Provider. The Docker daemon provided the required functionality to manage the container runtime and container image manipulation. This functionality was exposed to the web server via a Docker Remote API client library. The API endpoints accessible via HTTP request methods were developed as part of the web server functionality that consume (i.e. utilize) the Docker client library to support the various use cases of the Tool Client and the TP. The web server communicates with two different data stores: (1) the session and (2) the persistent storage for storing and retrieving user session information and storing &&  retrieving assignment configurations respectively. Figure 4.2 presents these components. Details of the web server are given in Section 4.1.2, while details of the Docker daemon's remote API are given in Section 4.1.3. The session storage is described in Section 4.1.4 and the Persistent Storage in Section 4.1.5.
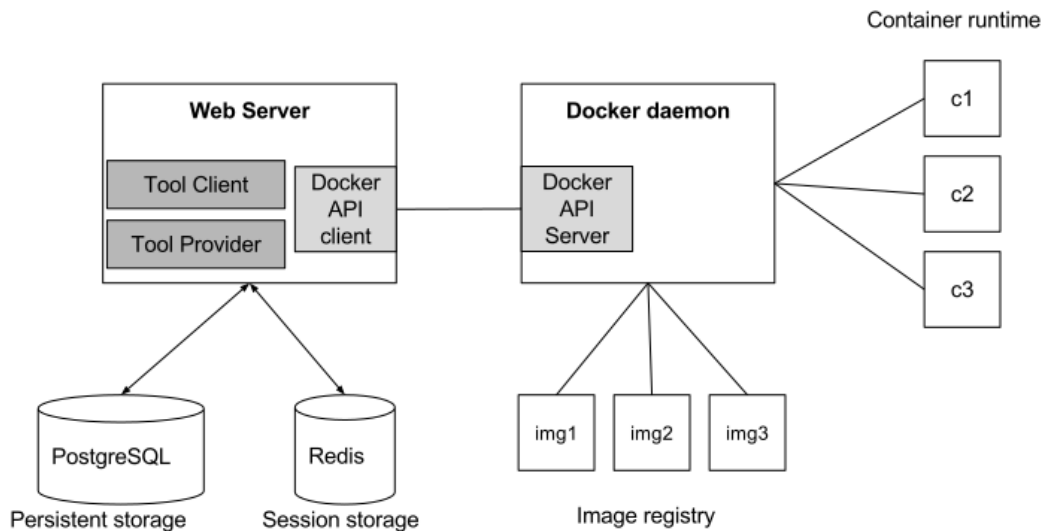


Figure 4.2: Architecture of the system components

## 4.1.1 Canvas LMS

Canvas LMS was used during the development phase of this project to understand and test the functionality of the LTI integration with the TP. Canvas is based on the Ruby on Rails framework [65] and has several software dependencies. To facilitate the installation of Canvas, a virtual machine was configured to run the Ubuntu 14.04 operating system. The software dependencies of Canvas were installed in the operating system as explained in the "Quick Start" wiki page of the official Canvas LMS GitHub repository[66]*.

---

*A simplified method for installing Canvas LMS in a virtual machine using Vagrant [67] and VirtualBox [68] was developed and used in this project. This method is documented in a picture

After the installation was complete and the system was running successfully, Canvas was configured to have an administrator account. This account was used to register two additional user roles (the *instructor* and the *student*), the institution (*KTH*), the department (*ICT*), and a course (*Internetworking*). The instructor user was configured to have the Canvas role *teacher* for this course, while the student user was configured to participate in this course. Figure 4.3 shows the configuration performed via the Canvas User Interface (UI).

(a) Structure of a course in Canvas

(b) People participating in a Canvas Course

Figure 4.3: Sample configuration of a course and its participants in Canvas LMS

The configuration of the LTI app and course assignments were previously explained in Figures 2.3 and 2.4 (respectively) of Section 2.4.1.

GitHub repository [69]. The README.md file is included in Appendix A.

### 4.1.2 Web server

The TP and Tool Client components are sets of API endpoints that are served by the same web server. Each endpoint is responsible for carrying out a specific task, such as authentication, exposing system resources to users, and launching LTI integrations. These endpoints were implemented using the Go programming language (also known as Go or Golang) [70, 71]. The web server itself, is implemented in Go and is part of the standard `net/http` [72] package. Listing 4.1 shows an example of an HTTP web server that is configured to listen for TLS connections on port 443 and has a single endpoint that replies to HTTP GET requests for the root (`"/"`) URL path.

Listing 4.1: Golang simple HTTPS web server

```go
import (
  "net/http"
  "github.com/julienschmidt/httprouter"
)

func handler(w http.ResponseWriter, req *http.Request, _
   httprouter.Params) {
  w.Header().Set("Content-Type", "text/plain")
  w.Write([]byte("This is an example server.\n"))
}

func main() {
  router := httprouter.New()
  router.GET("/", handler)
  http.ListenAndServeTLS(":443","cert.pem","key.pem",router)
}
```

The line `import "net/http"` includes the package which implements the HTTP web server. The line `import "github.com/julienschmidt/httprouter"` includes a Go package developed by Julien Schmidt [73], which maps URL paths such as the root path `"/"` to HTTP Request methods (such as GET, POST, DELETE, PUT, etc.), and Go functions such as `handler(w http.ResponseWriter, req *http.Request, _ httprouter.Params)` that process the corresponding HTTP request.

Go language is a strictly typed language, similar to C and C++. The declaration of variables, function parameters, and function return types is performed by first writing the corresponding parameter, variable or function name followed by its type. In the example above the function `handler` has three parameters `w`, `req`, and `_`. The type `http.ResponseWriter` is an interface that exposes functionality such as setting an HTTP response header and writing an HTTP response. The prefix `http.` indicates that `ResponseWriter` is a type that is part of the package `http`. Functions, types, and variables that are declared in a package and start with an uppercase letter, are exported by the compiler, hence are available for use in other packages, by first invoking the package name followed by a dot, and

then referring to the type, function, or variable. The type `http.Request` is an interface that exposes functionality for reading request parameters, form data, etc. The type `httprouter.Params` is a key-value data structure that maps http request parameters names to their values. Since such data are not relevant in the function, instead of naming the parameter, the blank identifier (represented by the underscore symbol) is used.

The server starts executing following the call to `ListenAndServeTLS(":443",` `"cert.pem", "key.pem", router)` function. This first parameter is the port number that the server will be listening on for incoming TLS connections, `cert.pem` and `key.pem` are the TLS certificate and key respectively that were generated similarly to the instructions in Section 2.4.2, while `router` is the URL router created in the line above it. The router parameter is declared and initialized using the symbols `:=`. This syntax tells the compiler to infer the type of the variable router from the return type (Router) of function `New()` that is declared in package `httprouter`. The call `router.GET`, takes two parameters, the URL path `"/"` and the function `handler`. The function GET registers the fact that every GET HTTP Method to the root path should be handled by function `handler`.

The implementation developed during this project, defines a series of functions (such as the handler function mentioned earlier) that implement functionality, such as creating a docker image, launching an assignment, etc. Each function is mapped to a specific URL path and an HTTP Method. Table 4.1 shows the URL Paths, the HTTP Method, and explains the functionality realized by each endpoint.

Table 4.1: Endpoints of the HTTP Web Server

| URL Path & HTTP Method | Endpoint functionality |
|---|---|
| /admin/login<br>POST | Implements the login functionality for the admin user of the LTI Tool Client |
| /admin/logout<br>GET | Implements the logout functionality for the admin user of the LTI Tool Client |
| /admin/containers/run/:id<br>POST | Handles the container run functionality for the admin user of the LTI Tool CLient. Parameter :id is the identifier of the image to be used for creating and starting a container. |
| /admin/containers/kill/:id<br>DELETE | Handles the container kill and remove functionality for the admin user of the LTI Tool CLient. Parameter :id is the identifier of the running container. |
| /admin/containers/commit/:id<br>POST | Handles the image creation functionality for the admin. It uses a specific running container as a seed for the new image. Parameter :id is the identifier of the running container. |
| /admin/images<br>GET | Lists all container images available to the admin user of the LTI Tool CLient. |
| /admin/images/history/:id<br>GET | Returns information about a particular image to the admin user of the LTI Tool CLient. Parameter :id is the identifier of the image. |
| /admin/images/delete/:id<br>DELETE | Deletes a particular image. Parameter :id is the identifier of the image. |
| /lti/launch/:id<br>POST | The LTI Tool Provider. Handles the LTI Launch request. Parameter :id is the identifier of the image that should be used to create and start a container for this particular request. |
| /ui/*filepath<br>GET | Handles requests for all static files that are located in a custom directory. The syntax of the URL route is related to the specification of httprouter package. |

### 4.1.3 Docker Remote API Consumer

The web server communicates with the Docker daemon via the Docker Remote API [46] to create and delete images and then, create, start, and delete docker containers. The web server uses the Go implementation of the Remote API Client library [74] to make requests to the Docker server. The version of the Docker Server used in this implementation is `1.12.4`, and the version of the server API was `1.24`. The version of the API is very important when initializing the client library from the Go code, as a matching version ensures that the client will communicate using the same version of the API calls that the server is responding to. This Remote API Client library has functionality similar to the Docker command line client that was introduced in Section 2.7. Listing 4.2 shows how a request is performed by the client `Cli` to start a container using the `ContainerStart` function of `Cli`. It is assumed that the container was previously created using the `ContainerCreate` function.

Listing 4.2: Start container request

```
Cli.ContainerStart(context.Background(), containerID, types.
    ContainerStartOptions{})
```

The first parameter expects a variable of type Context*, the second parameter is the unique identifier of the container to start. The last parameter is a Go struct of type `types.ContainerStartOptions` and its members are initialized with the *zero values* of their corresponding type using the curly brackets {}.† The `ContainerStartOptions` is part of the Docker Checkpoint & Restore [77] functionality that is not relevant to this project, hence no further explanation of it is given.

The functionality of the TP relies on facilitating a connection to a laboratory environment via the web shell emulator Shell in a Box. In order to support this functionality, a docker image with a pre-configured installation of Shell in a Box was chosen to serve as the initial container image of the system. The Tool Client allows the administrator to choose this initial image as a seed for creating new laboratory environments. The docker daemon stores images in its local image registry from various remote image repositories. The system was designed to access only a particular subset of images of the local image registry. Section 2.7 explained that a container image is identified by a repository, author, and a

---

*The package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes [75]. The background function returns a non-nil, empty Context. This context is never canceled, has no values, and has no deadline. The context is typically used by the main function, initialization, and tests, and as the top-level Context for incoming requests.

†The Go language specification [76] describes the initialization of variables as follows: When storage is allocated for a variable, either through a declaration or a call of new, or when a new value is created, either through a composite literal or a call of make, and no explicit initialization is provided, the variable or value is given a default value. Each element of such a variable or value is set to the zero value for its type: false for booleans, 0 for integers, 0.0 for floats, "" for strings, and nil for pointers, functions, interfaces, slices, channels, and maps.

version. In the Docker Remote API, the repository and the version of an image are identified by a parameter named `RepoTags`, i.e., an image of Ubuntu with version 14.04 has the RepoTag `ubuntu:14.04` where the semicolon is the delimiter between the repository and the version. The system is allowed to operate only on images that belong to a particular repository in order to satisfy the requirement for containers that can be accessed via a web shell emulator. In this implementation the repository was named `dc` and cannot be changed by any user of the system, while the image version identifies the different images and is a parameter that the administrator can set when a new image is created.

The source code of this implementation includes a Go package named `dc` (named after docker containers). This package is responsible for manipulating the images of the homonym repository, initializes the API client, and contains functions that consume the Docker API Client library. These functions are invoked by several HTTP route handlers in the Tool Client and the Tool Provider to deliver the desired functionality to the end users. The list below introduces the names of these functions along with brief descriptions of their intended functionality. This functionality is explained in more detail in the next sections of this chapter.

- `ListImages` requests the Docker API to return a list of all of container images, and afterwards, iterates over the results to filter out only images of the `dc` repository. This function is invoked by the endpoint `/admin/images`.

- `ImageHistory` requests the Docker API to return detailed information about a particular image (such as the author, the `RepoTags`, when was it created, and a text message that identifies the creation of the image). This function is invoked by the endpoint `/admin/images/history/:id`.

- `ImageRemove` requests the Docker API to remove a particular container image from the local repository. This function is invoked by the endpoint `/admin/images/delete/:id`.

- `RunContainer` first requests the Docker API to create a container from a specific image, and then starts the container. This endpoint returns configuration information for the user to access the container via the web SSH emulator. This function is invoked by the endpoint `/admin/containers/run/:id` and the endpoint `/lti/launch/:id`.

- `RemoveContainer` first requests the Docker API to stop a running container, and then to remove it from the container runtime (for example, this can be used after a container session expires for a user). This instance of the laboratory environment is purged and is no longer available for the system or the users. This function is invoked by the endpoint `/admin/containers/kill/:id`.

- `CommitContainer` requests the Docker API to create a new image using a running container as seed. For example, this can be used when an instructor

is running a container instance to configure software for a new laboratory environment. Once she is done with the configuration, she performs a request to "commit the container" as an image, in the local repository. This function is invoked by the endpoint `/admin/containers/commit/:id`

### 4.1.4 Session Storage

The system uses an in-memory key-value storage to store and retrieve information for user and container sessions. When a container is running for a particular user, whether that user is an administrator of the Tool Client, or a student who is accessing a laboratory environment via the LMS, the session storage stores information needed by the system to uniquely identify the user. In addition, the running container is stored in this storage. This mechanism prevents users from running multiple instances of a specific laboratory environment at the same time*, thus preventing resource exhaustion.

The session storage is realized by the open source in-memory data structure store Redis [78]. The server communicates with Redis using the client library for Go [79]. The information stored for a student session has the format `key-value`, where the key is a unique identifier for the user, while the value is a JSON object containing information about the running container. Every data entry has a Time To Live (TTL) value that defines when the key expires. For the system, an expired key means that the session has expired, thus a container should neither exist in a running state nor should the user be able to access it. The code sample in Listing 4.3 shows the value of a Redis key, used to identify a running container for an admin user of the Tool Client:

Listing 4.3: Redis session value for a container run configuration

```
{
  "id"       : "b79803d58414fd7786",
  "port"     : "4200",
  "username" : "admin",
  "password" : "password",
  "url"      : "https://localhost:4200"
}
```

The `id` is the identifier of the container. A Shell in a Box web server that is running in a container listens for connections on a specific port number. The attribute `port` is the port number that the host system is using to forward data packets to the port of the running container. The attributes `username` and `password` are additional parameters that the user should use to authenticate herself to access the emulated unix shell, and finally, `url` is the URL containing the hostname and the port to

---

*In this implementation, a user is limited to run only one laboratory environment at a time. This means that an instructor cannot access two different containers at a time and a student can only run and access one laboratory environment at a time. The implementation does not limit users from having multiple SSH sessions to the same container, hence a user can access the shell emulator multiple times from different browser tabs or windows.

access the shell emulator. For an admin user, such an entry has a key with a format such as `run:adm:7ff10abb653dead4186089acbd2b7891`, where `run:adm:` is the prefix, and `7ff10abb653dead4186089acbd2b7891` is a hash of the administrator's numeric account identifier. For a student the corresponding key has the format `run:usr:7272818191010`, where the prefix is `run:usr:` and `7272818191010` is the user identifier that is returned by Canvas via the LTI Launch integration.

Additional key-value data entries are stored in Redis, such as HTTP cookie information for users of the Tool Client. Such keys have the format `adm:7ff10abb653dead4186089acbd2b7891`, have a TTL of one day, and are created when the administrator successfully authenticates herself to the Tool Client.

### 4.1.5 Persistent Storage

The system uses a Relational Database Management System (RDBMS) to store persistent, information such as login credentials for the administrative user. The database server is PostgreSQL [80] with version number 9.6. A combination of two Go packages are used to establish connections, and then store and retrieve data from the PostgreSQL database. The first Go package is `database/sql` [81]. This package provides a generic interface interface to SQL databases. This package is intended to be used in conjunction with a database driver that implements the SQL interface functions. In this implementation the database driver is provided by the Go package `pq` [82].

Although the data stored in the persistent storage are not enough to justify the use of a RDBMS, a full-featured RDBMS was chosen to support future engineering choices that will extend the functionality of the system, such as storing information for assignments, and analytics regarding the usage of the system. This additional information will be available to the instructor via the Tool Client interface. This future work is documented in Section 5.2.

The current relational schema consists of a single table called `admins` that stores information such as the unique numeric identifier (`id`) of an admin user, the `username` and `password` that the administrator uses to sign into the Tool Client, a status that can be `active` or `deleted`, and optional information such as the `name` of the user, and timestamps that indicate when the admin account was created and when the user last signed into the Tool Client. Listing 4.4 presents the Structured Query Language (SQL) database schema definition using PostgreSQL specific syntax.

Listing 4.4: Relational Database Schema of the Tool Client

```
CREATE TYPE enum_admin_status AS ENUM('active', 'deleted');
CREATE TABLE admins(
  id SERIAL PRIMARY KEY,
  username varchar(60) NOT NULL UNIQUE,
  password varchar(100) NOT NULL,
  name varchar(100),
```

```
  status enum_admin_status NOT NULL DEFAULT 'active',
  created_at TIMESTAMP WITHOUT TIME ZONE DEFAULT
   CURRENT_TIMESTAMP ,
  last_login TIMESTAMP WITHOUT TIME ZONE
);
```

The method for accessing and storing data using the `lib/pq` package in Go is unimportant for this project, hence it has been left out. Moreover, the official documentation of the `pq` package [82] covers these methods in detail.

### 4.1.6 Binding network ports of the host system to container ports

The web server of the Tool Client and the TP is required to run multiple containers for several users at the same time. As explained in Section 2.8, each container is running the Shell In A Box web server process . In order for the shell emulator to be accessible from a user's browser, network packets from the host system should be forwarded to the corresponding docker container via a network bridge interface (docker0). This is achieved by binding TCP ports of the host system to the TCP port 4200 of each container running the shell emulator web server process. To avoid port collision on the host server, the system reserves and utilizes a specific port range between `4200-4399`. This means that the system has the ability to support a maximum of 200 running containers at the same time, hence, the web server can serve a maximum of 200 requests to run containers via the `/admin/containers/run:id` and `/lti/launch:id` endpoints.

Several mechanisms have been used to avoid port collision and to guarantee that the system has sufficient port resources to create new containers. During the startup process of the web server, a key-value data structure is initialized that stores the TCP port numbers as keys, while values are of boolean type and indicate whether the port is in use by a container or not. The definition of the data structure in Go code is:

```
type portResources struct {
  portsAvailable map[int]bool
}
```

The code states that `PortResources` is a struct that contains the map data structure `portsAvailable`.

When the function of the `dc` package `RunContainer executes` following a request to run a container for any of the `/admin/containers/run/:id` or `/lti/launch/:id` endpoints, the system will check if there is an available port in the map, and if so it will set this port's associated value to `true` to indicate that the port is in use. Similarly, when the function `RemoveContainer` is invoked, the system will locate the port in the map, and set its value to `false`, thus making the port reusable. This functionality covers the use cases when users manually request to run and kill containers.

Section 4.1.4 introduced the user sessions and their corresponding running configuration keys in Redis. As noted earlier these sessions are defined to expire

after some specific TTL. When a container session expires, the container is still running, but the key is removed in Redis. This indicates that the container should be terminated, and the port should become available for reuse by the system. A module named `PeriodicChecker` has been developed, that periodically checks whether the ports used by Docker containers are consistent with the `PortResources` map and the session keys in Redis. If for some reason a container is running and is using a port within the specified range, but the corresponding map entry does not have the value `true`, the mechanism will fix this inconsistency. Similarly, the system will check for inconsistencies in the Redis storage. If a key is missing for a container that is running, it assumes that the container has expired, and the container should be killed, and the port resources should be returned to the system for use. Algorithm 1 shows pseudocode that describes the the functionality of the `PeriodicChecker` module.

---

**Algorithm 1:** Module PeriodicChecker

$usedPorts := $ `getPortsOfDockerContainers()`
**foreach** $port \in PortResources$ **do**
    **if** $port \in usedPorts$ **then**
        $PortResources[port] := $ `true`
    **else**
        $PortResources[port] := $ `false`

**foreach** $port, containerID \in usedPorts$ **do**
    **if** $port \notin redisPorts$ **then**
        `RemoveContainer(`$containerID$`, `$port$`)`

---

The first line performs a series of calls to the Docker Remote API, to determine which containers are running in the system, and what host ports are used for these containers. The function returns the `usedPorts` map, with the ports as keys, while the values are the container identifiers. The following loop iterates over the `PortResources` map and resets its entries. A port entry of `PortResources` that exists in `usedPorts`, gets the value `true`, while an entry that does not exist in `usedPorts` gets the value `false`. Finally, the last loop, iterates over the entries of the `usedPorts` map, checks whether an entry for such a port exists in Redis storage, and if it does not, it invokes the `RemoveContainer` function of the `dc` package tp request via the Docker API to remove the container from the Docker runtime, and then releases the port from `PortResources` map by setting the corresponding entry to false.

Accessing the `PortResources` data structure is performing with the use of a *mutex* mechanism. Such a mechanism ensures that two functions that are running concurrently cannot access the data structure simultaneously, thus avoiding race conditions and enforcing atomic operations on the data structure. In Go, a function that can execute concurrently is called a *goroutine* [71]. Every *handler* function of the web server is executed as a *goroutine*. For this initial implementation, the go

package `sync` was used, which provides mutex locking functionality that blocks the execution of a goroutine when a mutex is locked. For example, when a function is trying to read the list of ports, it will try to lock the mutex. If a lock exists on that mutex, the function will stop executing, until the mutex is unlocked. This mechanism ensures atomic operations on the `PortResources`, thus avoiding two containers mapping to the same host port, and avoiding the `PeriodicChecker` manipulating the port resources that are accessed by another goroutine at the same time.

The `PeriodicChecker` was implemented to solve issues encountered during the development phase of the Tool Client. These issues were:

- Restarting the web server results to a new empty `PortResources` data structure, while containers are still running, and running container configurations exist in the Redis session storage. The mechanism described in this section ensures that the `PortResources` data structure will be filled with data associated with running containers during the start-up process of the web server.

- A container stops or crashes for some reason (i.e., the admin user created an image that broke the configuration of the Shell In A Box web server). The `PeriodicChecker` module will set the value of an unused port to `false`, and then remove any existing running configurations from the Redis session storage.

## 4.2    LTI Tool Client

The LTI Tool Client acts as an administration panel for the system. It allows a user with admin privileges* to create and delete docker images that act as pre-configured laboratory environments. These docker images are used when configuring an LTI integration for a course assignment in Canvas. This section explains the intended functionality of the Tool Client, presents the web pages of the Tool Client UI, and describes the key concepts used in this implementation.

### Authentication

The API endpoints consumed by the Tool Client UI have access restrictions to prevent unauthorized requests, i.e., requests not from an administrator. The process for authenticating an administrator and creating a user session is performed by submitting a web form with username and password parameters. This form is presented in Figure 4.4.

---

*A user with admin privileges is defined to be a user that has an entry in the table `admins` of the PostgreSQL database with the `status` column having the value `active`.

Figure 4.4: Signin form - LTI Tool Client Interface

When the *Sign in* button is clicked by the user, the Javascript function shown in Listing 4.5 executes to perform the following steps. First it accesses the form's parameters, then it sets the HTTP request URL to `/admin/login`, the header Content-Type to `application/json`, then it sets the HTTP request body to contain the following JSON object:

```
{
  "username": "admin",
  "password": "password"
}
```

Listing 4.5 shows the jQuery function `jQuery()`, which locates the HTML form using the HTML class attribute with value `form-signin`. The following function call `.submit` specifies which function will execute, when the submit button (Sign in) of the HTML form is clicked. The body of that function calls the jQuery function `$.ajax()` [83] which performs an AJAX request to the `/addmin/login` endpoint. The `$.ajax()` function has the parameters `url`, `type`, `dataType`, `data`, `contentType`, `success`, and `error`.

Listing 4.5: Javascript function consuming the /admin/login/ endpoint

```javascript
jQuery('.form-signin').submit(function() {
  $.ajax({
    url: "/admin/login",
    type: 'post',
    dataType: 'json',
    data: JSON.stringify({
      username: $("#userName").val(),
      password: $("#Password").val(),
    }),
    contentType: "application/json",
    success: function(data) {
      window.location.replace("/ui/images.html");
    },
    error: function(response) {
      // error handling
    }
  });
});
```

The parameter `url`, specifies the url path that is used to perform the HTTP request, and correspond to the server endpoint that handles the request. The parameter type, is the HTTP method to use for this request. The parameter `type` specifies the format of the data that is passed in the HTTP request body, and the parameter `data`, contains the JSON object shown earlier that is generated by locating the HTML input elements with identifiers `#userName` and `#Password` using the jQuery function `$()`*, and extracting their values by invoking the jQuery function `.val()`. These data are converted to a JSON object using the function `stringify()` of the Javascript object `JSON`. The parameter `contentType` sets the HTTP header to `application/json`, and is the HTTP request header that the server is expecting. The parameter `success` specifies the Javascript function to execute if the server responds with an HTTP StatusOK status code (200), while parameter `error` specifies the function to execute if the HTTP response contains a status code different than 200[†].

If the server replies that an error occurred, a corresponding error message is presented to the user, while if the request was successful, the user is redirected to the home page of the Tool Client interface.

The server validates the form data and compares the given parameters with the corresponding values of the user entry in the persistent storage. If the credentials match, a user session is created and stored in the session storage, and then an HTTP cookie is created containing information to uniquely identify this administrator. Afterwards, every subsequent request to other endpoints of the Tool Client, verifies that a cookie exists for that particular user, and that its value matches an existing

---

[*]The jQuery function `$()` is an alternative way of writing the function `jQuery()`, that accepts a string parameter.

[†]The HTTP response status codes are explained in detail by RFC 7231 [84].

entry in the session storage. If no such value exists either in the cookie value or in Redis session storage, then the user is redirected to the sign in form.

### Home Page - List of Images

The home page of the application is entitled "List of Images" (see Figure 4.5). This page contains a table with three columns: the image identifier (ImageID), the name of the image (Name), and the date the image was created (Created At). The user can click on each row of the table to go to the next page named "Image History" which provides more detailed information about this specific image.



Figure 4.5: Tool Client page "List of Images"

When the browser renders the HTML elements of this web page, it performs an HTTP GET request (as shown in Listing 4.6) for URL path /admin/images using the same $ajax() function that was presented earlier, with different parameters, specifically the GET HTTP method as type and /admin/images as url. The server upon successful authentication of the request, calls the dc function ListImages to request the Docker Remote API to return the list of images of the dc image repository, and afterwards, prepares a JSON array as a response, containing image information as a response. An example of the data in such a response is:

```
{
  "data": [{
    "Id": "00db67e76050",
    "RepoTags": "dc:0.1_traceroute"
    "CreatedAt": "2016-12-29T13:51:33+02:00"
  }, {
    "Id": "83364c85cafc",
    "RepoTags": "dc:0.0_seed"
    "CreatedAt": "2017-01-01T12:45:48+02:00"
  }]
}
```

If the server responds with HTTP status code 200, the jQuery function `$.each()` iterates over the JSON array contained in the response to parse the data and appends a row in the HTML table (using the jQuery function `append()`) for each array element.

Listing 4.6: Javascript function consuming the /admin/images/ endpoint

```
$(document).ready(function() {
  $.ajax({
    url: "/admin/images",
    success: function(response) {
      $.each(response.data, function(k, v) {
        $("#image-table").append(
          '<tr onclick=\'toImageHistory("' + v.Id + '")\' role
  ="button">'+
            '<td>' + v.Id + '</td>'+
            '<td>' + v.RepoTags + '</td>'+
            '<td>' + v.CreatedAt + '</td>'+
          '</tr>')
      });
    },
    error: function(response) {
      handleError(response)
    }
  });
});
```

The jQuery function `$(document).ready()` provides a way to run Javascript code, when the page's Document Object Model (DOM) becomes safe to be manipulated, and *before* the user can view or interact with the page content. When the "List Images" page is loaded, this function executes a call to the `$ajax()`. On `success`, it parses the `response` object provided by `$ajax`, and then iterates over the `response.data`. Function `function(k,v)`, specifies the action to be performed for each key (k), and value (v), of the JSON array. The call to `$("#image-table").append` locates the HTML table with the attribute identifier `#image-table`, and then appends a table row use the `<tr>` element. The attribute `onclick` specifies the action to be performed when the user clicks on a table row. This action is a call to a Javascript function named `toImageHistory()`, which requests the server to return the HTML page "Image History" that contains details about a particular docker image.

### Home Page - Image History

When the administrator clicks on a table row in the page "List of Images", an HTTP GET request to the `/admin/images/:id` endpoint is performed, similarly to the call presented in Listing 4.6. The server authenticates the request and afterwards it requests the Docker API to return information about a particular image. The server returns a JSON object with the requested information as shown below:

```
{
  "Id"         : "4165bca12451"
  "RepoTags"   : "dc:0.1_traceroute"
  "Comment"    : "Installed traceroute package"
  "Created At" : "2017-01-01T12:45:48+02:00"
}
```

This JSON object is parsed by the `success` function, and then its content is injected in the HTML page using the `$(#id).append()` function as shown earlier. The resulting page is shown in Figure 4.6.
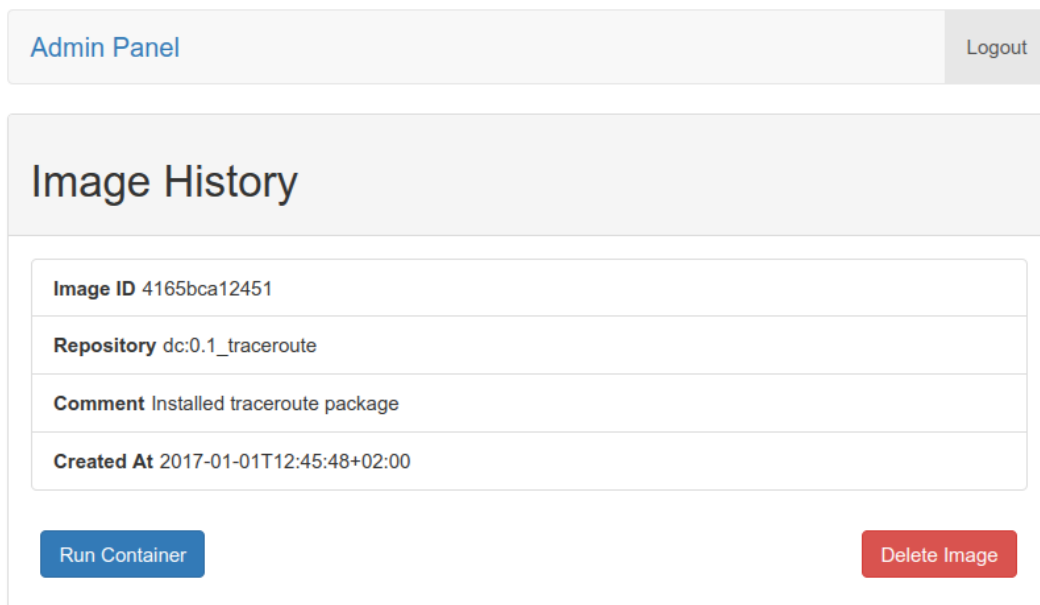


Figure 4.6: Tool Client page "Image History"

In addition to the information returned by the server, the user is presented with two button options: "Run Container" and "Delete Image". Run Container requests the page of the Tool Client that consumes the `/admin/containers/run/:id` endpoint, and gives the admin user access to this image as a running container, while Delete Image requests the Tool Client page that consumes the `/admin/images/delete/:id` endpoint and presents the admin user with an option to delete the container image from the system.

### Run Container page

The "Run Container" page works similarly to the previously explained pages. When the user clicks on the corresponding button of the page "Image History", before the HTML of the page is rendered by the browser and presented to the user, a request to the `/admin/containers/run/:id` endpoint is performed. The parameter `:id` is

the identifier of the image, that is used to create and start a container. The server performs the following steps following the POST request to the endpoint:

1. Verifies that the correct HTTP cookie was sent with the request. If the request is not authorized, an HTTP response with HTTP status code `401 Unauthorized` is returned.

2. Validates the request parameter `:id`. If the image identifier is not valid, then a response with HTTP status code `400 Bad Request` is returned.

3. Extracts the session key from Redis, and then looks for this container's run configuration. This check provides information for the endpoint handler, to know whether an existing container session exists and should be returned as a response or a new request should be made to the Docker API for running a container. This mechanism prevents subsequents requests from running new containers if a session already exists, thus preventing resource exhaustion for ports.

4. If a container session already exists, then the TTL value of the Redis key is renewed, and the JSON value of the key (shown in Listing 4.3) is returned as part of the HTTP response.

   If no container session was present in Redis storage, then a request is made to the Docker Remote API to run a new container. The server will first look for an unused port resource. If no ports are available, then an HTTP response with an error message is returned. If an unused port is found, it is reserved, and a container is created using the configuration parameters port, username, and password that are required for the Shell In A Box web server. The request for running the container is performed similarly to the example of the docker command `docker run` presented in Section 2.8. The major difference is that the web server performs two requests to the Docker Remote API via the Go client, by calling the functions `ContainerCreate` and `ContainerStart` of the client library.

   Finally, if the Docker API responds with success and starts the container, then a new JSON configuration entry is stored in Redis and this JSON entry is returned as a response to the calling jQuery function.

Figure 4.7 presents the contents of the web page "Run Container". The page contains an an HTML iframe, that embeds the Shell In A Box shell emulator, with an active SSH session. The page shows the user which credentials they need to use to login into the Linux shell (Username and Password), and below the iframe are two buttons: "Commit Container" and "Delete Container". When these buttons are clicked a request is made to the corresponding web pages of the Tool Client (the first is responsible for creating an image from a running container, while the second is responsible deleting the running container).
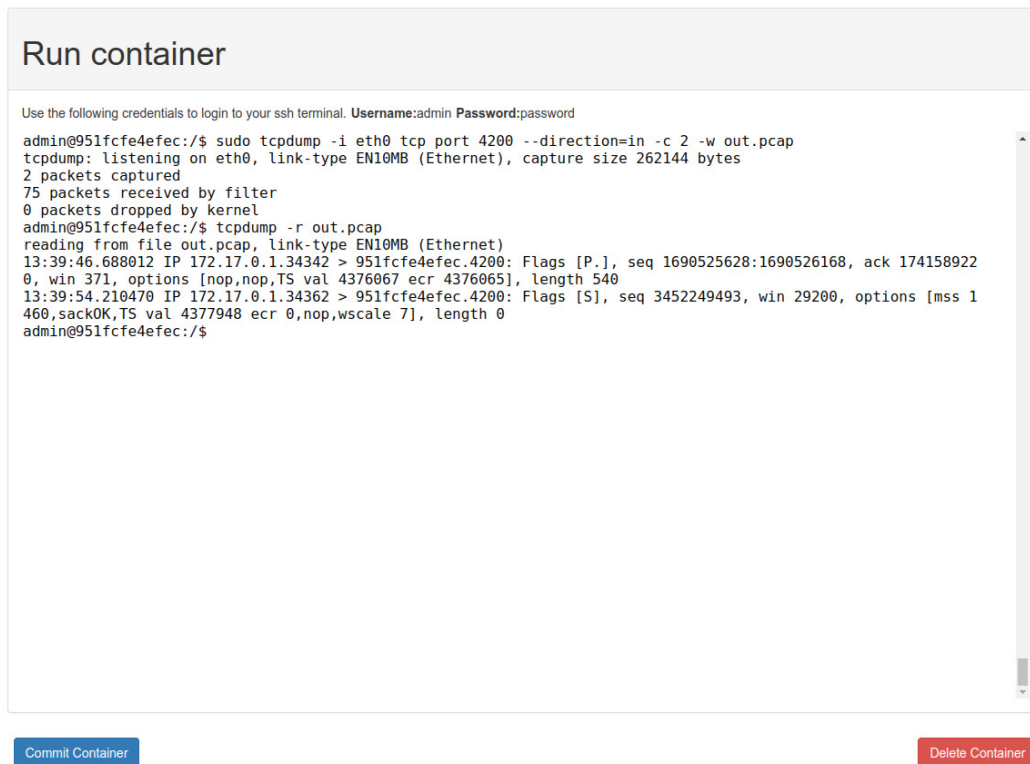
Figure 4.7: Tool Client page "Run Container"

The contents of the terminal presented in Figure 4.7* show the administrator user (named "admin") running the `tcpdump` program [85] to perform a capture of incoming TCP packets to the network interface `eth0` and the port 4200. The program captures packets sent from the Shell In A Box emulator while the user is typing commands in the terminal. The packets are forwarded to the web server that is running inside the container. This web server is listening for connections on port 4200. There are two commands shown in the figure. The first is `tcpdump -i eth0 tcp port 4200 --direction=in -c 2 -w out.pcap`. The parameter `-i` specifies the network interface to listen on, in this case `eth0`. The parameter `tcp` specifies to listen only for TCP packets. The parameter `port` specifies the destination port on which the targeted TCP packets will arrive. The parameter `--direction=in` specifies to listen only for incoming TCP packets. The parameter `-c` specifies that the program should stop listening after capturing the first 2 packets. Finally, the parameter `-w` specifies the output file into which tcpdump should write the results. The second command `tcpdump -r out.pcap` has the parameter `-r` which specifies the input file from which it should read the previously captured output. When the

---

*The user named "admin" has successfully logged in the shell using the given username and password, and then has executed the command `clear` to remove any previous output from the terminal. These two steps have not been included in Figure 4.7.
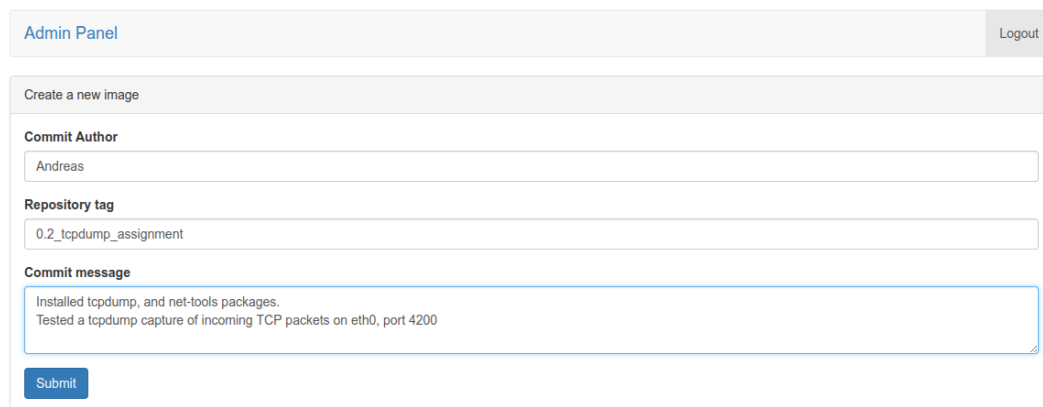
program runs it will output information about the previously captured packets on the standard output.

This process can be used by an instructor to specify an assignment that involves the use of `tcpdump` program. The instructor can configure the required software for the assignment, and then test the assignment inside the emulator to verify that the laboratory environment is configured correctly and is ready to be used by students. Once the configuration is complete, the admin user of the Tool Client clicks on the "Commit Container" button to visit the corresponding page in order to create a container image (as described in the next section), to subsequently be used for configuring an LTI integration in Canvas.

### 4.2.1 Commit Container page

The "Commit Container" page allows the admin user of the Tool Client to create new images using a running container as its configuration. The page (shown in Figure 4.8) contains an HTML form with three input fields to be used as metadata when storing the image in the local Docker image repository. The first field is Commit Author, i.e., the name of the author (i.e., user) who issues the commit command. The second is Repository Tag. The value of this field will be used to identify the image. The last input field is Commit message. This field allows the admin user to provide additional information for the image. Examples of this input data were presented earlier on the "Image History" page (shown in Figure 4.6). The form has a submit button, that when clicked performs an HTTP POST request to the `/admin/containers/commit/:id` endpoint. The Javascript function that parses the form data and performs the request is similar to that of the "Login page" (see Listing 4.5).



Figure 4.8: Tool Client page "Commit Container"

The handler function of the endpoint authenticates the user's request, then performs validation of the form's input fields, and then requests the Docker Remote API to create a new container image. This is performed using the

function `ContainerCommit` of the Go client library. If the image is successfully created, the server issues a request to delete the container and its corresponding and port mappings from Redis session storage.

### 4.2.2 Delete Container page

The "Delete Container" page (shown in Figure 4.9) presents the user with a message to confirm that they wish to delete a running container, and a button labeled "Delete Container". When this button is clicked, a Javascript function is triggered to perform an HTTP DELETE request to the `/admin/containers/kill/:id` endpoint to delete the container. The handler function of the endpoint authenticates the user's request, verifies that the container is actually running by checking for this container's running configurations in Redis. Finally, the handler requests the Docker Remote API to remove the container using the function `ContainerRemove` of the Go client library, and afterwards, all related session keys are removed from the Redis session storage.
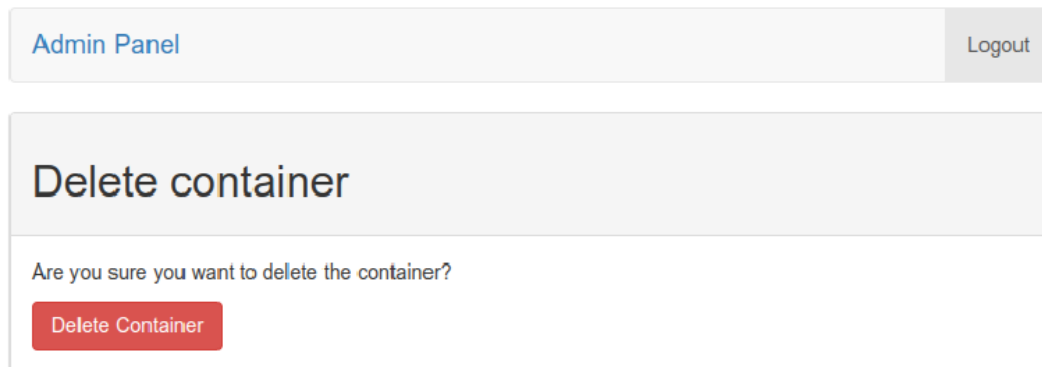


Figure 4.9: Tool Client page "Delete Container"

### 4.2.3 Delete Image page

The "Delete Image" page (shown in Figure 4.10) is loaded after clicking the corresponding button in the "Image History" page. It works similarly to the "Delete Container page", but performs an HTTP DELETE request to the `/admin/images/delete/:id` endpoint. The handler of the endpoint authenticates the user's request, performs validation of the image identifier, and requests the Docker Remote API to delete the image by calling the function `ImageRemove` of the Go client library.
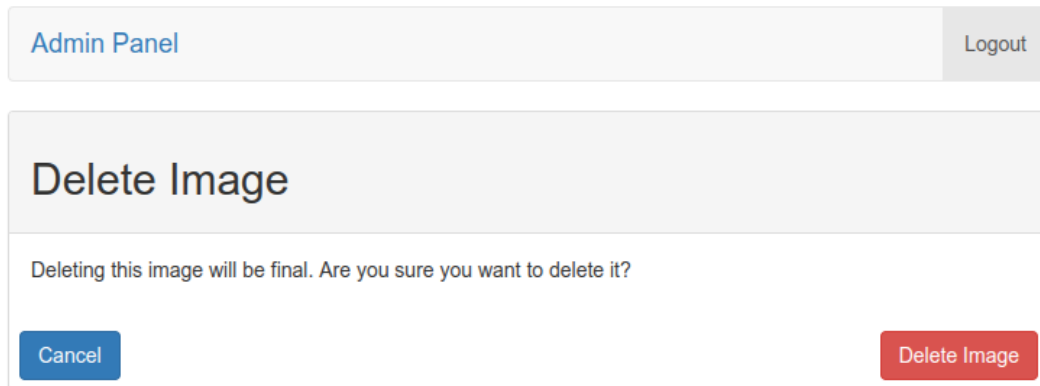
Figure 4.10: Tool Client page "Delete Image"

## 4.3 LTI Tool Provider

The TP is realized by a single endpoint called `/lti/launch/:id`. The parameter `:id` is used to identify the container image that should be used to run a container. The handler function of the endpoint has a mechanism for authenticating requests from Canvas LMS in a similar way as the Sinatra web application presented in Section 2.4. This mechanism is implemented using a Go library for LTI integrations[86]. The route definition is:

```
router.POST("/lti/launch/:id", route.OAuth(route.LTILaunch))
```

The HTTP method is POST, while the handler functions that serve the request for the URL `/lti/launch/:id` are `OAuth`, and `LTILaunch`. The first is the authentication mechanism and its implementation is shown in Listing 4.7, while the latter is the function that serves requests for running laboratory environments.

The `OAuth function` is of type `httprouter.Handle` (this type is defined as `type Handle func(http.ResponseWriter, *http.Request, Params)`), has the parameter `handler` that is a function of the same type, and has the same return type. Several programming languages including Go support passing functions as arguments to other functions or specifying them as return values. Such languages are often categorized as programming languages with support for "First Class Functions". In the listing below, the OAuth function, defines the return function right after the reserved word `return`. This function is responsible for authenticating requests for the LTI route.

The authentication is performed using the OAuth 1.0 Protocol. A request to this route is expected to have an OAuth signature that matches a predefined key and a secret. Such a signature is sent by Canvas. The signature is based on the key and secret values defined during the integration of an external application (such as the TP). When Canvas performs the LTI Launch request, it sends an OAuth signature. The server verifies that signature as shown in the code in Listing 4.7.

Listing 4.7: Authentication of the LTI Launch requests in Go

```go
func OAuth(handler httprouter.Handle) httprouter.Handle {
  return func(res http.ResponseWriter, req *http.Request,
   params httprouter.Params) {

    // OAuth authentication of the TP requires to match the
    // request URL to match the expected path. Since image IDs
    // change all the time, the path is constructed using
    // the imageID as extracted from the HTTP Header.
    path := fmt.Sprintf("https://%s%s",req.Host,req.URL.Path)

    p := lti.NewProvider("oauth_secret", path)
    p.ConsumerKey = "oauth_key"

    ok, err := p.IsValid(req)
    if !ok {
      res.Write([]byte("Invalid request"))
      return
    }
    if err != nil {
      res.Write([]byte("An error occured"))
      return
    }
    handler(res, req, params)
  }
}
```

The parameter `req` contains the signature value and method that are sent by the LMS. The OAuth signature provider is created following a call to the function `NewProvider()`, which takes two arguments `oauth_secret` (the secret that protects the route) and path (the URL path of the route). The key of the TC is configured by the assignment `p.ConsumerKey = "oauth_key"`. The call to the function `IsValid(req)` creates a server-side signature and compares it against the signature sent by the TC. This function has two return values, a boolean `ok` and an error. If the result contains an error, or `ok` does not have the value `true`, then error messages are returned in the HTTP response. If the signature matches, then the `handler` function is invoked to create a new laboratory environment.

The `handler` function `LTILaunch` operates similarly to the function that handles requests to the `/admin/containers/launch/:id` endpoint (as previously explained in Section 4.2), but instead of returning a JSON object as a response, the LTILaunch handler returns an HTML page containing the credentials for logging into the shell together with an iframe with the shell emulator embedded in it. The resulting page is shown in Figure 4.13, while a simplified version of the handler's code is shown in Listing 4.8.

Listing 4.8: LTILaunch route handler function

```go
func LTILaunch(res http.ResponseWriter, req *http.Request,
   params httprouter.Params) {

  t, _ := template.ParseFiles("templ/html/assignment.html")

  // Validate imageID
  if !vImageID.MatchString(imageID) {
    t.Execute(res, Resp{Error: "Invalid URL. Contact the
   administrator"})
  }

  // Parse LTI Post params
  err := req.ParseForm()
  // Error handling is omitted in listing

  // extract Canvas userID and store is as session key
  userID := req.PostFormValue("user_id")
  sessionExists, err = dc.ExistsUserRunConfig(userID)
  // Error handling is omitted in listing

  if sessionExists {
    cfg, err = dc.GetUserRunConfig(userID)
    // Update the TTL
    err = dc.SetUserRunConfig(userID, cfg)
  } else {
    // SESSION didn'texist, Generate username and password
    username := "guest"
    password := newPassword()

    // Run container request
    cfg, err = dc.RunContainer(imageID, username, password)
  }

  // Set session
  err = dc.SetUserRunConfig(userID, cfg)

  // Return HTML template with data
  t.Execute(res, getResp(cfg))
}

type Resp struct {
  ContainerID  string
  Port         string
  Username     string
  Password     string
  URL          string
  Error        string
}
```

The first action of the LTILauch function is to create a text template following a call to function `ParseFiles()` of Go package `html/template`. The function takes an HTML file as an argument and produces a variable of type `Template`. The function `t.Execute(res, RespError: "text message")` is used to inject string values into the template `t` and to write the output to the <mark>HTTTP</mark> response `res`. The Go `struct Resp` contains values of type `string` that are used in the template to inject the URL of the iframe containing the shell emulator, the username, password, and container identifier. For example, a variable containing an error is passed in the HTML template using the `.Error` syntax. The `Execute` function will replace the contents of `.Error` with the value of the `Error` variable.

```
<span>Error:</span> {{ .Error }}
```

After the template variable is initialized, the handler validates the image identifier parameter `:id` via a call to `vImageID.MatchString(imageID)`. The variable `vImageID` is a compiled regular expression defined as `var vImageID = regexp.MustCompile(^([A-Fa-f0-9]12,64)$)`. The function `MatchString` verifies whether the parameter imageID of type string, matches the regular expression (an alphanumeric sequence of 12-64 characters consisting of a hexadecimal encoding of the container's identifier) and returns a boolean value as a result.

Afterwards, the handler reads the `user_id` form parameter sent by Canvas, and checks whether a container run configuration exists in the Redis session storage for that particular user. If such a configuration exists, then it is loaded in the `cfg` variable and the TTL value of the Redis entry for this configuration is renewed. If such a configuration was not present, then a `username`, and a random `password` are created and passed as parameters to the function `RunContainer` of package `dc` to create and start a new container for this user session. The new container run configuration is stored in Redis following a call to `SetUserRunConfig`.

Finally, a call to `t.Execute(res, getResp(cfg))` is performed, to write the configuration values into the HTML template and to return these values to the invoking Canvas LMS. The function `getResp(cfg)` initializes a `Resp` struct with the values returned from the `RunContainer` function.

Section 4.3.1 contains an example of configuring an `/lti/launch/:id` route as an external application in Canvas LMS and Section 4.3.2 contains an example of a student accessing a laboratory environment through an assignment that was configured to launch the external application.

## 4.3.1 Configuration of an Assignment

Figure 4.11 shows how a specific image was configured in Canvas, as an external application. The name of the application is `tcpdump_01`, the consumer key and the shared secret have values `oauth_key` and `oauth_secret` (respectively). These values must match those configured in the OAuth handler function (shown in Listing 4.7). The Launch URL is `https://localhost:8080/lti/launch/9f6ffc322b08` where

the identifier of the container image is the identifier created in the Tool Client from the "Commit Container" page of Figure 4.8.



Figure 4.11: Configuration of the TP in Canvas

An assignment configuration was created in Canvas to run the external tool shown above. The tool was instructed to run in a new browser window, rather than embed the response of the TP in the same page. For the configuration of the assignment, the laboratory assignment "Hands-on 6: Understanding TCP and tcpdump"[1] from the course "6.033: Computer System Engineering" of the Massachusetts Institute of Technology (MIT), "Electrical Engineering &

Computer Science Department" was used. A description of this assignment is shown in Figure 4.12.

In this assignment you will understand how TCP works using tcpdump.

In your home directory you will find a file named **tcpdump.dat** .

For this trace, we used a program that transmits a file from a machine called **willow** to a machine called **maple** over a TCP connection. We ran the **tcpdump** tool on the sender, willow, to log both the departing data packets and the received acknowledgments (ACKs).

The file **tcpdump.dat** is a binary file which contains a log of all the TCP packets for the above TCP connection. The file is not human-readable. To understand the log file in a human-readable format, run:

```
tcpdump -r tcpdump.dat > outfile.txt
```

Now open outfile.txt on your preferred text editor. The output has several lines listing packets sent from willow to maple, and the ACKs from maple to willow. For example:

```
00:34:41.474225 IP willow.csail.mit.edu.39675 > maple.csail.mit.edu.5001: Flags [.], seq 1473:2921, ack 1, win 115, options [nop,nop,TS va
l 282136474 ecr 282202089], length 1448
```

Denotes a packet sent from willow to maple. The time stamp 00:34:41.474225 denotes the time at which the packet was transmitted by willow.

TCP uses sequence numbers to keep track of how much data it has sent. For teaching purposes, we often associated one sequence number with each packet (packet 1, packet 2, etc.). In reality, there is one sequence number per *byte of data*. The above packet has a sequence number 1473:2921, indicating that it contains all bytes from byte #1473 to byte #2920 (= 2921 - 1) in the stream, which is a total of 1448 bytes.

(**Note:** There may be very minor variations in the format of the output of tcpdump depending on the version of tcpdump on your machine.)

Once maple receives the packet, assuming that it has received all previous packets as well, it sends an acknowledgment (ACK):

```
00:34:41.482047 IP maple.csail.mit.edu.5001 > willow.csail.mit.edu.39675: Flags [.], ack 2921, win 159, options [nop,nop,TS val 282202095
 ecr 282136474], length 0
```

Again, for teaching purposes, we typically talk about an ACK reflecting the corresponding packet's sequence number. In reality, the ACK reflects the next byte that the receiver expects. The above ACK indicates that maple has received all bytes from byte #0 to byte #2920. The next byte that maple expects is byte #2921. The time stamp 00:34:41.482047, denotes the time at which the ACK was received by willow.

Questions:

1. What are the IP addresses of maple and willow on this network? (Hint: Check the man page of tcpdump to discover how you can obtain the IP addresses)
2. A TCP connection runs not just between two machines, but between two specific *ports* on those machines. What ports are used in the connection between willow and maple?
3. How many kilobytes were transferred during this TCP session, and how long did it last? Based on these numbers, what is the throughput (in KiloBytes/sec) of this TCP flow between willow and maple?
4. What is the round-trip time (RTT) in seconds, between willow and maple, based on packet 1473:2921 and its acknowledgment? Look at outfile.txt and find the round-trip time of packet 13057:14505. Why are the two values different?

This tool needs to be loaded in a new browser window

Load Sample tcpdump assignment in a new window

Figure 4.12: Assignment Description that could be placed into the Canvas LMS course (based upon the first part of the assignment in [1] - this material appears here based upon CC BY 3.0 US)

The description of Figure 4.12 instructs the user to use the `tcpdump` command line program that is pre-configured in the laboratory environment to study TCP packets that were sent from a server called `willow` to a server called `maple`. It provides some information regarding the output of `tcpdump`, and then asks the student a series of four questions to complete the assignment. Methods for replying to such questions are not presented in this example, as they are not relevant to the use of the laboratory environment. At the bottom of the assignment, an HTML button with content "Load Sample tcpdump assignment in a new window" is visible. When this button is clicked, Canvas performs the HTTP POST request to the `/lti/launch/:id` endpoint, and requests the browser to render the HTML response in a new window (shown in Figure 4.13).

### 4.3.2 Student accessing a laboratory environment

Figure 4.13 shows a student accessing the laboratory environment via the HTML page returned as a response from the `LTILaunch` route handler. The student has already authenticated herself in the shell, and is following the assignment's instructions to execute the command `tcpdump -r tpdump.dat > outfile.txt` to write the output of the TCP packet trace in a human readable format into file `outfile.txt`.



> **Laboratory environment**
>
> Use the credentials listed below to login to your ssh terminal. **Username:** canvas **Password:** 66yKff
>
> ```
> root@e379d843ae44:/# tcpdump -r tcpdump.dat > outfile.txt
> reading from file tcpdump.dat, link-type EN10MB (Ethernet)
> root@e379d843ae44:/# vi outfile.txt
> ```
>
> ❶ If this is your first session you might have to visit https://127.0.0.1:4212 in a new tab and accept the SSL certificate.
>
> Terminate container: Terminate

Figure 4.13: Laboratory environment via Canvas LMS

The bottom of the page contains a text (in blue background) that instructs the user to open the shell emulator in a new browser window, to leverage full screen capabilities of the shell. Finally, a button called "Terminate" provides the option for the user to terminate the container session.

## 4.4 Evaluation

The evaluation of the LTI Tool Client and Tool Provider implementations was performed using unit and integration testing techniques [87], to verify the functional correctness of the software against desired specifications. While the unit tests were developed to evaluate the individual components (units) of the system, such as the

validation of user input for all api requests, the configuration tool, and the port resource manager, the integration tests were developed to check the behavior of the LTI Tool Client and the Tool Provider as a system that interacts with its dependent services such as the Docker daemon, the Redis session storage and the PostgreSQL RDBMS.

The software artifact of this project was developed using the Clean Architecture software application design practice [88, 89]. The Clean Architecture models systems in four layers, structured as concentric circles, where the inner layers represent the domain model (i.e. docker images and containers) and their use cases (i.e. create a docker image from a running container), while the outer layers represent mechanisms for realizing the use cases (i.e. the API services and the communication with the docker daemon). Figure 4.14 shows the four layers of this architecture design model.
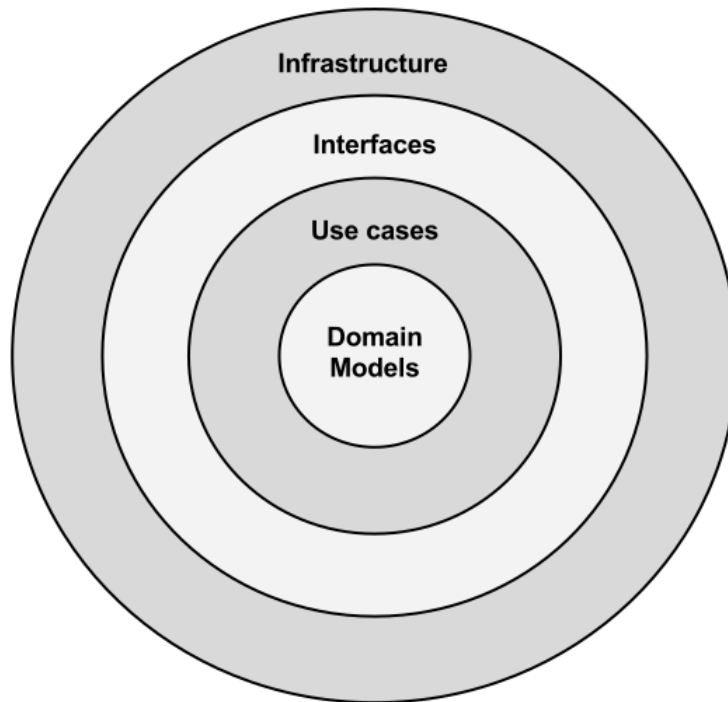


Figure 4.14: The four layers of the Clean Architecture

The `domain models` of this project are listed in Table 4.2.

Table 4.2: List of implemented domain models

| Model | Description |
|---|---|
| `Container` | A docker container |
| `Image` | A docker image |
| `Admin` | An administrator user of the Tool Client |
| `RunConfig` | A configuration of a running container that exposes an SSH session over an HTTP connection. |

Among several `Use Cases` the most notable are:

- Get all docker images

- Run a docker container

- Kill a docker container

- Commit a docker container to create a new image

- Find an administrator user by their username

The `Interfaces` represent interactions with the `Use cases` and domain models. The API routes introduced in table 4.1 are the web interfaces for exposing the use cases via the Restful API. Finally the connection with the docker daemon, the PostgreSQL RDBMS and the Redis key-value storage and the web server represent the `Infrastructure` layer of the architecture.

The clean architecture introduces an important rule which dictates that source code software dependencies can only point inwards (i.e. an `interface` for retrieving an admin object from the database, depends on a predefined admin `domain model`). An inner circle is never aware of software defined in an outer circle, thus simplifying dependency injection, and resulting to decoupled software packages that can be easily tested. In Go, declarations of such layers are realized by interfaces, that are named collections of struct methods having the purpose of contract that dictates the desired outcome of each member function. There can be multiple implementations for such interfaces. An example is an interface that describes a `SELECT` statement for a particular table in some RDBMS, can be implemented several times, using different database drivers. This is particularly useful when an underlying technology of the infrastructure layer changes, only a particular layer of the architecture will be affected. Such methodology of multiple implementations of the same interface is particularly useful when performing unit tests. For example, the business logic can be tested in forms of unit tests without a real database connection, by mocking the implementation of that inner layer (i.e providing an alternative implementation that simulates a connection with the database). As a result, each layer can be tested independent of the other layers, while the system as a whole can be tested using other techniques, such as integration tests.

Figure 4.15 shows the go packages and their structure, as implemented in this project. The directory `cmd` contains the source code for the HTTP web server (the main function). The rest of the source code is organized under the directory `pkg`. The package `pkg/drivers` contains the infrastructure layer that models the connections with the database (`docker`), the connection with the session storage (`redis`), and the docker API client library (`docker`). The package `repositories` contains the `interface` layer that implements the use cases for interacting with the domain model, and the directories under `pkg/api` contain the interfaces that implement the web services `container`, `image`, `lti` and `auth`, that provide the use cases for interacting with docker containers, provide authentication and authorization to the LTI Tool Client and Provider, etc. The directory `portmapper` implements the port management software, `config` provides functions for reading configuration parameters required by other packages, and lastly, the directory `integration` provides functionality for testing the system as a whole.

Figure 4.15: The source code directory tree of this project

```
cmd
└── dock_server
pkg
├── api
│   ├── auth
│   ├── container
│   ├── image
│   ├── lti
│   ├── portmapper
│   └── repositories
├── config
├── drivers
│   ├── docker
│   ├── postgres
│   └── redis
integration
```
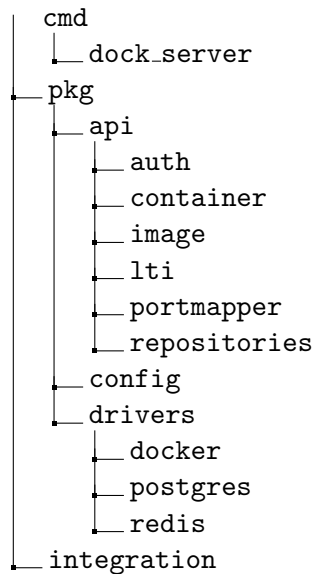
The next sections explain how unit tests and integration tests are performed in Go. Section 4.4.1 shows how the source code of this project was tested using unit testing. Section 4.4.2 shows how test coverage of source code is calculated in Go and introduces the test coverage report for this project. Section 4.4.3 shows how the system as a whole was evaluated using integration testing and introduces benchmarks for each API endpoint.

### 4.4.1 Unit testing in Go

The Go programming language includes the package `testing`[90] that provides functionality for testing individual units (functions) of a program. A test is a function that its name contains the prefix `Test`, accepts a single argument that is a pointer to the `*testing.T` data structure and resides in a file suffixed with `_test.go`. The data structure `T`, provides functions to terminate a test given a failure. A test is executed by running the command `go test`. Listing 4.9 shows an example of a simple test for a function `Sum` that calculates the sum of two integers.

Listing 4.9: Example of a simple unit test in Go

```go
package sum_test

import "testing"

func Sum(x,y int) int { return x+y }

func TestSum(t *testing.T) {
  s := Sum(1,1)
  if s != 2 {
    t.Errorf("Incorrect sum, expected: %d", actual: %d, 2, s)
  }
}
```

The test runs by executing the command `go test -v sum_test.go`. The flag `v` requests for verbose output for each test of the file.

```
go test -v sum_test.go
=== RUN   TestSum
--- PASS: TestSum (0.00s)
PASS
ok      command-line-arguments   0.001s
```

The output indicates that the go test command line tool executed one test which succeeded (indicated as `PASS`) and in total it took `0.001` seconds.

Most tests written for this project use additional packages that provide functionality for performing assertions which are easier to read, have better output for errors, and execute parallel tests for permutations of the functions' input arguments. Listing 4.10 shows an example of the `TestSum` in such format.

Listing 4.10: Example of a simple unit test in Go

```go
import (
  "testing"
  "github.com/stretchr/testify/assert"
)

func TestSum(t *testing.T) {
  tests := []struct {
    x, y, expect int
    name         string
  }{
    {
      x:      1,
      y:      1,
      expect: 2,
      name:   "Good test",
    },
    {
      x:      1,
      y:      5,
      expect: 3,
      name:   "Force an error",
    },
  }
  for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
      actual := Sum(tt.x, tt.y)
      assert.Equal(t, tt.expect, actual)
    })
  }
}
```

The slice `tests` defines a list of struct objects that contain three fields, `x,y,expect` that correspond to the two input integers of function `Sum` and the expected output, and the field `name` that defines a name for each test permutation. The permutations are looped using the keyword `range`, and each permutation is assigned to the variable `tt`. The function `Run` of package `testing` allows to run nested tests within the `TestSum` function. Such tests will run in parallel, and inform the parent test about their success or failure. The package `assert` provides various functions for performing assertions that improve readability of the source code and test output. In this example the function `Equal` takes three arguments, the pointer `t` to the `T` data structure, the expected output `expect` of the function `Sum` and the `actual` output. The second permutation is written with incorrect expected output to show how failure error messages are presented to the user.

```
go test -v sum_test.go
=== RUN    TestSum
=== RUN    TestSum/Good_test
=== RUN    TestSum/Force_an_error
--- FAIL: TestSum (0.00s)
  --- PASS: TestSum/Good_test (0.00s)
  --- FAIL: TestSum/Force_an_error (0.00s)
  Error Trace:    sum_test.go:35
  Error:     Not equal: 3 (expected)
             != 6 (actual)

FAIL
exit status 1
FAIL    command-line-arguments   0.002s
```

After executing this test, the standard output shows that `TestSum` executed two tests where the `TestSum/Good_test` suceeded while `TestSum/Force_an_error` failed because the expected output did not match the actual output.

Listing 4.11 shows the implementation of a function that initializes a connection with the Docker daemon using the Docker API client library, as explained in Section 4.1.3.

Listing 4.11: Source code of the Docker API client

```go
package docker

import (
  "os"
  "github.com/docker/docker/client"
)

// APIClient encapsulates the Docker Remote API client
type APIClient struct {
  Cli *client.Client
}

// NewAPIClient initializes a new Docker API client.
func NewAPIClient(dockerConfig map[string]string) (*APIClient,
    error) {
  _ = os.Setenv("DOCKER_API_VERSION", dockerConfig["version"])
  _ = os.Setenv("DOCKER_HOST", dockerConfig["host"])
  cli, err := client.NewEnvClient()
  if err != nil {
    return nil, err
  }
  return &APIClient{Cli: cli}, nil
}
```

The function `NewAPIClient()` initializes the connection with the Docker daemon. It accepts a single argument `dockerConfig` that is of type `map[string]string`, that contains the version of the Docker API and the host url where the daemon is responding to requests (either a unix socket or an HTTP connection URL). It sets two environment variables `DOCKER_API_VERSION` and `DOCKER_HOST` and performs a call to the function `NewEnvClient()` of the `client` package of go library for that implements the docker remote API. If `NewEnvClient()` fails to initiate a connection it will return an error, and a nil pointer to struct `APIClient`.

Listing 4.12 shows how this code is tested from the function `TestNewAPIClient`.

Listing 4.12: Unit test of intializing a connection with the Docker API

```
package docker

import (
  "testing"
  "github.com/stretchr/testify/assert"
)

func TestNewAPIClient(t *testing.T) {
  cli, err := NewAPIClient(map[string]string{
    "version": "1.25",
    "host":    "unix:///var/run/docker.sock",
  })
  assert.NoError(t, err)
  assert.NotNil(t, cli)
  cli2, err := NewAPIClient(map[string]string{
    "version": "x",
    "host":    "local",
  })
  assert.Error(t, err)
  assert.Nil(t, cli2)
}
```

First it provides valid argument to the function `NewAPIClient`. It performs two assertions, one that the function did not return an error by calling `assert.NoError(t, err)`. This function checks whether the error variable is `nil` or not. Then it performs the second assertion that the client is not nil by calling the `assert.NotNil(t, cli)` function and passing the client as an argument. Similarly, the second test case, provides invalid argument to `NewAPIClient`, forcing the docker Remote API Client library to produce an error, thus returning a nil pointer of type `APIClient` and an error.

**Mocking dependencies in unit tests**

The test of Listing 4.12 assumed that a docker daemon was running at the time the test was executing. If the daemon was not running, both test cases would have failed, and the test would not be successful. There are cases in which running actual software dependencies such as the Docker daemon, or a Redis server might be inconvenient, or outside the scope of a unit test. For example, the route handler of the `Logout` API endpoint of the LTI Tool Client, attempts to match a valid session cookie in the HTTP request with an entry in the Redis session storage. If such entry exists in the storage, it will attempt to delete it. When testing the individual function `Logout`, validation of the request cookie is of higher importance than potential connection errors of the Redis client. Since the system as a whole is tested using integration tests, the dependency of the connection with Redis will

be simulated (mocked), and only the functional correctness of the Logout function against various inputs will be evaluated.

Listing 4.13 shows the source code of the Logout HTTP handler. First it reads the cookie with name `ses` from the HTTP request. If it fails, a call to function `api.WriteErrorResponse` is performed, with a status code `http.StatusUnauthorized` that corresponds to http status code 401 as defined in section 3.1 of RFC 7235 of the Hypertext Transfer Protocol (HTTP/1.1), and a corresponding error message string `"Unauthorized"`. Then it will attempt to delete the corresponding entry in the Redis session storage. In case of failure it returns an error, and in case of success, it invalidates the cookie and writes it to the response HTTP headers, along with an empty response body (call to function `api.WriteOKResponse()`).

Listing 4.13: Source code of Admin Logout HTTP handler

```go
// AdminLogout logs out an admin
func (s Service) AdminLogout(w http.ResponseWriter, r *http.
   Request, _ httprouter.Params) {

  // Get session cookie
  cookie, err := r.Cookie("ses")
  if err != nil {
    api.WriteErrorResponse(w, http.StatusUnauthorized, "
   Unauthorized")
    return
  }

  // Check if session exists in Redis. If it doesn't exist
   sent Unauthorized. Frontend will redirect to login page.
  if err = s.redis.AdminSessionDelete(cookie.Value); err !=
   nil {
    api.WriteErrorResponse(w, http.StatusInternalServerError,
   err.Error())
    return
  }

  cookie = &http.Cookie{
    Name:    "ses",
    Value:   "",
    Path:    "/",
    Expires: time.Now(),
  }
  http.SetCookie(w, cookie)
  api.WriteOKResponse(w, nil)
}
```

The code above shows that the struct `service` has a field `s.redis` which is a golang interface, of type `RedisRepository`. `RedisRepository` is an implementation of the Interfaces layer of the architecture, that uses the redis

driver from Infrastructure layer and provides a function `AdminSessionDelete` that deletes a key from Redis. Listing 4.14 shows the implementation of the repository and its `AdminSessionDelete` function.

Listing 4.14: Sample of the Redis repository interface and its implementation

```go
// The RedisRepository interface with a method signature
   called AdminSessionDelete
type RedisRepository interface {
  AdminSessionDelete(key string) error
}

// The RedisRepo implements the RedisRepository interface
type RedisRepo struct {
  redis redis.Redis
}
// AdminSessionDelete implements the method of the
   RedisRepository interface
func (r *RedisRepo) AdminSessionDelete(key string) error {
  _, err := r.redis.Del(key)
  return err
}
```

When testing the `Logout` route handler a connection to a Redis server is not available. Instead, an alternative implementation of the RedisRepository interface is initialized, and provided as argument to the `service` struct. That implementation returns either an `error` or `nil` when the test needs to test any of those cases. Listing 4.15 shows the test code of the route handler `AdminLogout`, the different permutations of the HTTP request input, and the alternative implementation of the RedisRepository from a package named `repomocks`.

Listing 4.15: Unit test of Admin Logout HTTP handler

```go
func TestAdminLogout(t *testing.T) {
  tests := []struct {
    service     Service
    request     *http.Request
    expectCode  int
    expectCookie *http.Cookie
    name        string
  }{
    {
      service:      NewService(nil),
      request:      httptest.NewRequest(http.MethodGet, "/",
  nil),
      expectCode:   http.StatusUnauthorized,
      expectCookie: nil,
      name:         "session does not exist",
    },
    {
      service: NewService(repomocks.NewRedisRepositoryMock().
          WithAdminSessionDelete(errors.New("redis network
  error"))),
      request:      cookieRequest("1"),
      expectCode:   http.StatusInternalServerError,
      expectCookie: nil,
      name:         "deleting session errors",
    },
    {
      service: NewService(repomocks.NewRedisRepositoryMock().
          WithAdminSessionDelete(nil)),
      request:      cookieRequest("1"),
      expectCode:   http.StatusOK,
      expectCookie: &http.Cookie{Name: "ses", Value: "", Path:
  "/", Expires: time.Now()},
      name:         "deleting session succeeds",
    },
  }
  for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
      w := httptest.NewRecorder()
      tt.service.AdminLogout(w, tt.request, nil)
      assert.Equal(t, tt.expectCode, w.Code)
      if tt.expectCookie != nil {
        assert.Equal(t, tt.expectCookie.String(), w.Header().
  Get("Set-Cookie"))
      }
    })
  }
}
```

The mocked dependency allows to test the functional correctness of the `AdminLogout` route handler without a real connection to a Redis server. The function call to `NewService` intializes a `Service` struct and takes the `RedisRepository` interface as an argument. The first test case, does not provide a valid request cookie, therefore the function will exit before the call to `AdminSessionDelete`. The second test case evaluates the response when the `AdminSessionDelete` function returns an error, while the last test case evaluates a valid request with a valid cookie and no errors returned by `AdminSessionDelete`, and the response code and cookie matches the expected code and cookie as defined in the test case.

### 4.4.2 Generating a test coverage report

The `go test` command allows for generating a coverage report. Test coverage is a term that describes how much of a package's code is exercised by running the package's tests [91]. If a test invokes x% of a package's source statements, then that package has x% test coverage. The coverage module instruments * the binary source code using the GNU's Not Unix! (GNU) `gcov`[93] tool by adding break points to every branch, and calculating if those breakpoints are reached during a an invocation of a test. The number of covered branches over the total breakpoints produces the coverage percentage. The `go test` command allows to write all coverage statistics into a single file called the *profile* for further analysis.

The listing below shows how to invoke the `go test` command using the `cover` flag to generate a coverage report for the source code of Listing 4.11 that was presented in the previous section.

```
go test -v -cover -coverprofile=coverage.txt -covermode=count
    github.com/andreas-kokkalis/dock_server/pkg/drivers/docker
=== RUN   TestNewAPIClient
--- PASS: TestNewAPIClient (0.00s)
PASS
coverage: 100.0% of statements
ok    command-line-arguments   0.004s
```

The flag `covermode` with value `count` indicates that it mould't only check if a statement run, but also how many times it run. The flag `coverprofile` indicates to write the coverage statistics into the output file `coverage.txt`.

```
mode: count
pkg/drivers/docker/docker.go:15.71,19.16 4 2
pkg/drivers/docker/docker.go:22.2,22.34 1 1
pkg/drivers/docker/docker.go:19.16,21.3 1 1
```

where the output numbers match the following fields: `name.go:line.column,line.column numberOfStatements count`.

---

*Instrumentation[92] is a source code insertion technology that adds specific code to the source files under analysis. After compilation, execution of the code produces dump data for runtime analysis or component testing.

The field `name.go` indicates the filename, the first occurrence of `ling.column`. Opening this file using the `go tool cover -html=coverage.txt` command line tool, will open a web browser window such as the one of Figure 4.16 to show a human readable view of the coverage report. Statements highlighted with green color are covered fully, statements highlighted with red are not covered, while statements in gray are slightly covered.
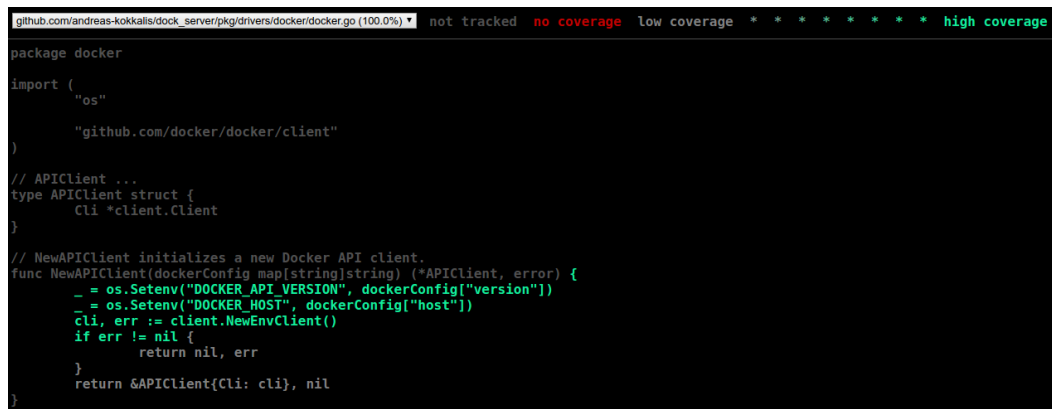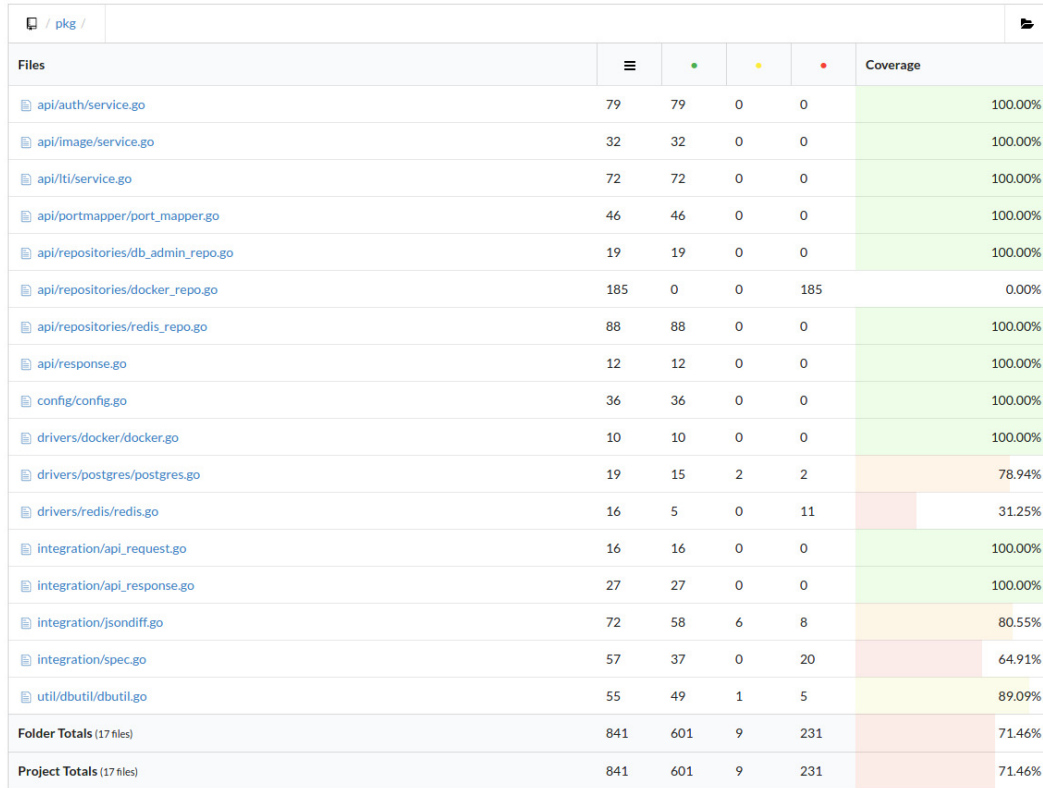


Figure 4.16: Sample of the go tool cover HTML output

This project uses the Travis [94] Continuous Integration (CI) [95] tool to run its tests. Travis provides for free, resources that enable to run programs under various configurable conditions. This project's Travis configuration enables to compile the source code using go version 1.9 on an Ubuntu 14.04 virtual machine, that runs a docker daemon. The development environment is completely reproducible in that remote virtual machine, where the reader can investigate the various builds and tests that were implemented and executed as part of this project [96, 97].

During a Travis build, the following events are taking place. First the code is compiled, along with all its software dependencies. Then the unit tests are executed, and then the integration tests run. If any of the unit or integration tests fail, the build is considered unsuccessful. Finally, the unit tests run again to generate coverage reports, and upload them to Code Coverage[98] service that analyzes test coverage reports from various programming languages and provides historical data and statistics about the tests of a particular project. Figure 4.17 shows the overview of coverage percentage per file, as exported from codecov.io [99] for the git commit number `a794b854eddea7fe8556c7897d51d57aa08fecc5` of `master` branch of the GitHub repository [97] of this project.

| Files | ≡ | ● | ● | ● | Coverage |
|---|---|---|---|---|---|
| api/auth/service.go | 79 | 79 | 0 | 0 | 100.00% |
| api/image/service.go | 32 | 32 | 0 | 0 | 100.00% |
| api/lti/service.go | 72 | 72 | 0 | 0 | 100.00% |
| api/portmapper/port_mapper.go | 46 | 46 | 0 | 0 | 100.00% |
| api/repositories/db_admin_repo.go | 19 | 19 | 0 | 0 | 100.00% |
| api/repositories/docker_repo.go | 185 | 0 | 0 | 185 | 0.00% |
| api/repositories/redis_repo.go | 88 | 88 | 0 | 0 | 100.00% |
| api/response.go | 12 | 12 | 0 | 0 | 100.00% |
| config/config.go | 36 | 36 | 0 | 0 | 100.00% |
| drivers/docker/docker.go | 10 | 10 | 0 | 0 | 100.00% |
| drivers/postgres/postgres.go | 19 | 15 | 2 | 2 | 78.94% |
| drivers/redis/redis.go | 16 | 5 | 0 | 11 | 31.25% |
| integration/api_request.go | 16 | 16 | 0 | 0 | 100.00% |
| integration/api_response.go | 27 | 27 | 0 | 0 | 100.00% |
| integration/jsondiff.go | 72 | 58 | 6 | 8 | 80.55% |
| integration/spec.go | 57 | 37 | 0 | 20 | 64.91% |
| util/dbutil/dbutil.go | 55 | 49 | 1 | 5 | 89.09% |
| **Folder Totals** (17 files) | 841 | 601 | 9 | 231 | 71.46% |
| **Project Totals** (17 files) | 841 | 601 | 9 | 231 | 71.46% |

Figure 4.17: Overview of project's test coverage report from codecov.io

The first column contains the name of the file, the second column the total lines of code per file. The third , forth and fifth columns show the number of lines fully, partially not not covered respectively, while the last column shows a summary percentage of the files statements coverage from tests.

### 4.4.3 Integration tests

Section 4.4.1 introduced how unit tests are developed in Go, and how this project utilized such tests to verify the functional correctness of the project. In most cases, individual layers of the system where tested, while dependencies were simulated by alternative implementations of the interfaces. Testing a system along with its dependencies is also very important, to verify that multiple units, when working with each other, can deliver the desired functionality according to some specifications. In the scope of this project, an integration test assumes that before testing some code, all its software dependencies are running correctly, i.e., the HTTP web server for the API, the Docker daemon, and finally the Redis and PostgreSQL servers.

The Ginkgo [100] Behavior Driven Development (BDD) [101] testing framework was used to define specifications for integration tests. BDD uses a form of natural language constructs to describe a software specification (testing suite) along with

its acceptance criteria under various conditions. The Ginkgo framework consumes the `testing` package of Go in order to run tests using the `go test` command line tool, and provides such natural language constructs via functions calls.

The developer must import two packages, `ginkgo` and `gomega`, where the first provides an API for natural language constructs while the second an API for performing assertions. The method of importing these packages is called dot imports, and allows for accessing functions of those packages directly, without using the package name prefix. The listing below shows how those imports are defined.

```
import (
  "testing"

  . "github.com/onsi/ginkgo"
  . "github.com/onsi/gomega"
)
```

All test specifications are executed from a test function by performing the following two calls:

```
func TestImageEndpoints(t *testing.T) {
  RegisterFailHandler(Fail)
  RunSpecs(t, "Image Suite")
}
```

The call to `RegisterFailHandler(Fail)` defines the behavior of the ginkgo framework when encountering a failure. This handler defines what messages are printed during a failure in the standard output, as well as if the framework will continue running tests succeeding a failed scenario. The call to `RunSpecs(t, "name of specification")`, indicates that ginkgo should execute all registered specifications in the test file. The pointer to the `testing.T` data structure is given as argument to `RunSpecs` to allow ginkgo to use the underlying functionality of the `testubg` package. All specifications for a given test are defined within the `Describe` function block as showed in Listing 4.16.

Listing 4.16: Structure of Gingo test specifications

```
var _ = Describe("Image api endpoints", func() {
  It("Should list all images", func() {
    // perform request
    // record response
    // assertions
  })
  It("Should get image history of seed image", func() {
    // ...
  })
})
```

The `Describe` function accepts a string parameter to indicate the name of the top level specification, and a function that is executed when invoking the `Describe`

function. The value of `Describe` is assigned to an unnamed variable. This ensures that Ginkgo will execute the specifications defined within that code block when the `RunSpecs` function is called. Within a `Describe` block, other functions named `It` are defined to describe sections of the testing suite that should occur at a sequential order. In the example of Listing 4.16 two sections are defined, the first to execute tests for the route `/admin/images` introduced in Table 4.1, while the second for the route `/admin/images/history:id`. The first argument of each of the `It` functions is a string variable that verbosely declares the intention of each section in a human readable way.

In addition to the `Describe` and `It` functions, Ginkgo provides other functions such as the `BeforeSuite` to define a series of actions that should be taken before executing any specifications, `AfterSuite` to define the actions to perform after executing all specifications, and `BeforeEach` and `AfterEach` that define actions to be taken before executing each `It` section of the specification. The latter is useful, for example when a test depends on having a particular state of the session or the persistent storage. Listing 4.17 shows an example of a `BeforeSuite` block that initializes all dependences of the `ImageService` that models all routes relevant to docker images.

Listing 4.17: Initial configuration of integration tests for the image routes

```
var _ = BeforeSuite(func() {

  spec = integration.NewSpec(dir)

  Describe("Initialize configuration", func(){
    c, err := config.NewConfig(path.Join(s.TopDir, confDir),
    environment)
    gomega.Expect(err).To(gomega.BeNil(), "Init config")
    spec.Config = c
  })

  Describe("Connect to redis", func(){
    redis, err := redis.NewClient(spec.Config.GetRedisConfig()
    )
    gomega.Expect(err).To(gomega.BeNil(), "Connect Redis")
    spec.Redis = redis
    spec.RedisRepo = repositories.NewRedisRepo(redis)
  })

  Describe("Connect to docker daemon", func(){
    dockerClient, err := docker.NewAPIClient(spec.Config.
    GetDockerConfig())
    gomega.Expect(err).To(gomega.BeNil(), "Init docker api
    client")
    spec.DockerCLI = dockerClient
    spec.DockerRepo = repositories.NewDockerRepository(
    dockerClient, spec.Config.GetDockerConfig())
  })

  Describe("Initializes HTTP routes", func(){
    router := httprouter.New()
    imageService := image.NewService(spec.RedisRepo, spec.
    DockerRepo)
    router.GET("/admin/images", imageService.ListImages)
    router.GET("/admin/images/history/:id", imageService.
    GetImageHistory)
    spec.Handler = router
  })
})
```

Within the `BeforeSuite` multiple `Describe` functions can be defined, to group various actions that need to be performed before executing any tests. The call to `integration.NewSpec` function initializes a data structure that holds information such as the connection to PostgreSQL, Redis, Docker daemon, the routes, as well as a set of member functions that perform HTTP requests, record the HTTP responses, and compare the response against an expected response. Each `Describe` block contains a call to initializing a connection to an infrastructure

layer resource (i.e., Redis, Docker, HTTP server) and an initialization of each repository of the `interface` layer of the Clean Architecture. The calls to `gomega.Expect()` functions declare assertions that are performed during the initialization of each service. If any of these assertions fails, the rest of the tests will not be executed, and the specification is considered unsuccessful.

Listing 4.18 shows the definitions for testing the two endpoints `GET /admin/images` and `GET /admin/images/history:id`. The first part of the specification, initializes a `Request` object that is defined in Listing 4.19 and is part of the `integration` package, by calling the function `NewRequest` and passing as arguments three parameters. The first (`http.MethodGet`) is the HTTP method to use, the second `"/admin/images` is the request URI that matches the registered route, while the last is an `interface{}` to any JSON serializable data structure to be used as the HTTP request body (in this example no JSON body is required, therefore the value of the parameter is `nil`). Then the specification initializes an object of the `Response` data structure that models an expected API response for a particular HTTP request. Listing 4.20 shows the definition of the `Response` data structure that is also part of the `integration` package, and the function `NewResponse` that initializes a pointer to a `Response` object. This function takes two arguments, the first is the expected HTTP status code that the server will respond, and the expected JSON body that the server will send as part of the HTTP response.

Listing 4.18: Performing a request to an endpoint within a spec file

```
var _ = Describe("Image api endpoints", func() {
  var img api.Img
  It("Should list all images", func() {
    request := integration.NewRequest(http.MethodGet, "/admin/
    images", nil)
    response := integration.NewResponse(http.StatusOK, imgspec
    .ImageListGood)
    spec.AssertAPICall(request, response)

    var images []api.Img
    response.Unmarshall(&images)
    img = images[len(images)-1]

  })
  It("Should get image history of seed image", func() {
    request := integration.NewRequest(http.MethodGet, fmt.
    Sprintf("/admin/images/history/%s", img.ID), nil)
    response := integration.NewResponse(http.StatusOK, imgspec
    .ImageHistoryGood)
    spec.AssertAPICall(request, response)
  })
})
```

Then, the function `AssertAPICall` that is shown in Listing 4.21 is executed to test

whether the actual server response matches the expected response. The `Response` structure provides some additional functions such as `Unmarshall` that allows for reading the actual JSON response into a Go data structure. This is particularly useful for capturing data that are required to be given as input to subsequent API requests. For example the call to `/admin/images/history/:id` route requires a valid docker image identifier, that is loaded from the array of images returned by the `/admin/images` endpoint.

Listing 4.19: An HTTP request as modeled in the integration package

```
// Request struct for performing an HTTP request
type Request struct {
  method string
  url    string
  body   interface{}
  // HTTPRequest models the HTTP request. It's exported allow
  for setting custon request headers and cookies.
  HTTPRequest *http.Request
}
// NewRequest intializes a Request object
func NewRequest(method, url string, body interface{}) *Request
    {
  jsonBody, err := json.Marshal(body)
  gomega.Expect(err).To(gomega.BeNil(), "Error marshaling body
    parameter to json")
  return &Request{
    method:       method,
    url:          url,
    body:         body,
    HTTPRequest: httptest.NewRequest(method, url,     ioutil.
   NopCloser(bytes.NewReader(jsonBody))),
  }
}
```

The HTTPRequest field of the `Request` struct is initialized using the `httptest` package of Go, by calling the `httptest.NewRequest` function. The parameters of this function are similar to the parameters of the `integration.NewRequest` function, with the slight difference that the body of `httptest.NewRequest` must be of type `io.Reader`, that models an input stream of data. The JSON serializable data structure `body` is first converted to json binary data using the function `json.Marshal`, and then converted into a stream by calling the function `ioutil.NopCloser(bytes.NewReader(jsonBody))`.

Listing 4.20: An expected HTTP response as modeled in the integration package

```go
// Response struct for asserting an http API response
type Response struct {
  expectedCode int
  expectedBody string
  recorder     *httptest.ResponseRecorder
}
// NewResponse initializes a Response object hat is used to
   test the expected output against the actual HTTP response
func NewResponse(expectedCode int, expectedJSONBody string) *
   Response {
  return &Response{
    expectedCode: expectedCode,
    expectedBody: expectedJSONBody,
    recorder:     httptest.NewRecorder(),
  }
}
```

The `Response` data structure holds information for an expected API response.
The `expectedBody` should be a valid `JSON` string and the `expectedCode` is the
HTTP status code that shall be returned after performing a particular HTTP
request. Function `NewResponse` initializes the `Response` object and returns a
pointer to that object. The field `recorder` is initialized following a call to
`httptest.NewRecorder` that returns a pointer to a `httptest.ResponseRecorder`
structure that implements the `http.ResponseWriter` interface which is used when
defining the route handlers for each endpoint. The recorder offers functionality for
retrieving the HTTP response headers and body and is used by the function
`AssertAPICall` to test the actual HTTP response against the expected one.

Listing 4.21 shows the implementation of `AssertAPICall` function that is also
part of the integration package. The function takes two parameters, a pointer to a
`Request` and to a `Response` object. First it initializes a timer by calling
`time.Now()` of the `time` Go package that is used to count the time that elapsed
while serving a particular HTTP request. The timer is stopped by calling
`time.Since(start)` that returns an integer that represents the amount of
nanoseconds that elapsed since the `start`. The call to
`s.Handler.ServeHTTP(response.recorder, request.HTTPRequest)` is calling
the function `ServeHTTP` of the `router` that was assigned to the field `Handler` of
the structure `Spec` as shown in Listing 4.17. The `ServeHTTP` function takes two
arguments, the `httptest.ResponseRecorder` and the `http.Request`. It matches
the URL of the request with one of the registered routes, and invokes the
corresponding route handler function. It records the response into the
`response.recorder` object. Afterwards the function writes to `stdout` the original
request, and actual response into a human readable format by calling the
`.pretty()` functions respectively). The next step is to start performing assertions.
The first assertion will check whether the actual HTTP status code of the
response matches the expected one. Then it will attempt to load the JSON

response body into the `api.Response` data structure, and perform an assertion to test if an error occurred. Finally it compares the actual response of the API endpoint with the expected JSON response. This step is performed by calling the `CompareRegexJSON` function of the integration package. This function actually executes a Python command line program called `json-regex-difftool`[102], an open source project licenced under the Apache Licence 2.0 [103] developed by ©Bazaarvoice Inc that allows to compare two JSON files to check if the key value pairs of each JSON object match. In addition, it offers to define an expected JSON value for an object, using a regular expression, which is useful when we want to evaluate a docker container or image identifier, that is generated during an HTTP request to one of the API endpoints, and the exact character sequence of the identifier is not known.

Listing 4.21: Assertion of an HTTP response of the integration package

```go
func (s *Spec) AssertAPICall(request *Request, response *
    Response) {

  // Perform HTTP Request
  start := time.Now()
  s.Handler.ServeHTTP(response.recorder, request.HTTPRequest)
  took := time.Since(start)

  // Log request and response to stdout
  s.Log.Printf("%s\n", request.pretty())
  s.Log.Printf("%s\n", response.pretty())
  s.Log.Printf("Took: %s\n", took.String())

  // Perform assertions
  gomega.Expect(response.Code()).To(gomega.Equal(response.
    expectedCode), "status codes do not match")

  var actualResponse api.Response
  err := json.Unmarshal(response.recorder.Body.Bytes(), &
    actualResponse)
  gomega.Expect(err).To(gomega.BeNil())

  diff, err := CompareRegexJSON(response.expectedBody,
    response.ToString(), s.TopDir)
  gomega.Expect(err).To(gomega.BeNil(), "Diff tool returned
    error")
  gomega.Expect(diff).To(gomega.Equal(""), "Diff is not empty"
    )
}
```

The call to `CompareRegexJSON` returns a string of the diff if the expected response does not match the actual, and an error, in case an error occurs while executing the command line tool. Both of these return values are evaluated in corresponding assertions to test that no error was returned, and that there was no

difference between the two JSON strings.

Listing 4.22 shows how an expected JSON response is defined using regular expressions for values of particular fields. The code of this listing represents a response of the `GET /admin/images` endpoint. The key `Id` is expected to have an alphanumeric sequence of 12 to 64 characters as value, while the key `CreatedAt` is expected to match anything as a value, since the time and date of the image creation is not of high importance for the assertion.

Listing 4.22: Example of a JSON expected response containing regular expressions

```
{
  "data": [
    {
      "Id": "([A-Fa-f0-9]{12,64})$",
      "RepoTags": [
      "andreaskokkalis/dc:0.2_tcpdump_assignment"
      ],
      "CreatedAt": "(.+)"
    },
    {
      "Id": "([A-Fa-f0-9]{12,64})$",
      "RepoTags": [
      "andreaskokkalis/dc:0.1_traceroute"
      ],
      "CreatedAt": "(.+)"
    },
    {
      "Id": "([A-Fa-f0-9]{12,64})$",
      "RepoTags": [
      "andreaskokkalis/dc:0.0_seed"
      ],
      "CreatedAt": "(.+)"
    }
  ]
}
```

The executing a Ginkgo specification that was described in this section is performed by invoking the `go test` command line tool as shown in the listing below.

```
go test -v ./pkg/api/image/spec -ginkgo.v
```

The flag `ginkgo.v` is passed to the specification, to indicate that it should print a verbose output of the tests. Listing 4.23 includes the output of the Ginkgo suite for the two image endpoints `/admin/images` and `/admin/images/history:id`.

Listing 4.23: Sample output of a successful Ginkgo integration test

```
=== RUN   TestImageEndpoints
Running Suite: Image Suite
==========================
Random Seed: 1515276041
Will run 2 of 2 specs

Image api endpoints
Should list all images
/home/andreas/workspace/golang/src/github.com/andreas-kokkalis
    /dock_server/pkg/api/image/spec/ginkgo_image_test.go:61

------------------
{
  "HTTP_Request": {
    "Method": "GET",
    "URL": "/admin/images"
  }
}
------------------
{
  "HTTP_Response": {
    "Code": 200,
    "Headers": {
      "Content-Type": [
      "application/json; charset=UTF-8"
      ]
    },
    "Body": {
      "data": [
      {
        "CreatedAt": "2017-01-04T05:53:37-05:00",
        "Id": "02ba2aacd9c9",
        "RepoTags": [
        "andreaskokkalis/dc:0.2_tcpdump_assignment"
        ]
      },
      {
        "CreatedAt": "2017-01-04T05:51:10-05:00",
        "Id": "976f25c6d342",
        "RepoTags": [
        "andreaskokkalis/dc:0.1_traceroute"
        ]
      },
      {
        "CreatedAt": "2016-12-29T06:51:33-05:00",
        "Id": "83364c85cafc",
        "RepoTags": [
        "andreaskokkalis/dc:0.0_seed"
```

```
        ]
      }
    ]
  }
}
------------------
Took: 14.527027ms


Image api endpoints
Should get image history of seed image
/home/andreas/workspace/golang/src/github.com/andreas-kokkalis
   /dock_server/pkg/api/image/spec/ginkgo_image_test.go:71

------------------
{
  "HTTP_Request": {
    "Method": "GET",
    "URL": "/admin/images/history/83364c85cafc"
  }
}
------------------
{
  "HTTP_Response": {
    "Code": 200,
    "Headers": {
      "Content-Type": [
      "application/json; charset=UTF-8"
      ]
    },
    "Body": {
      "data": [
      {
        "Comment": "",
        "CreatedAt": "2016-12-29T06:51:33-05:00",
        "CreatedBy": "",
        "Id": "83364c85cafc",
        "RepoTags": [
        "andreaskokkalis/dc:0.0_seed"
        ],
        "Size": 0
      }
      ]
    }
  }
}
------------------
Took: 887.08μs
```

```
Ran 2 of 2 Specs in 0.094 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 0 Skipped ---
    PASS: TestImageEndpoints (0.09s)
PASS
ok    github.com/andreas-kokkalis/dock_server/pkg/api/image/
    spec  0.101s
```

The output of a test contains the custom messages printed by the `AssertAPICall` function such as the HTTP request, the response and time it took to process it, and also information for each test, such as the string variables defined in each `Describe` and `It` block. In addition, the file and line number that each `It` block is defined is printed, along with a message for success (`SUCCESS`) or failure. At the end of the specification, it prints a summary for all tests that run, along with information for the total execution time of the tests.

### 4.4.4 Summary of tests

The initial implementation of the source code, was not performed using the Clean Architecture design model. Therefore the evaluation of the code using unit and integration tests became very inefficient. A refactoring for the source code was performed to apply such architecture, and start testing each component and layer of the system independently, using unit and integration tests. Both types of tests improved the quality and readability of the code, and resulted to the discovery of multiple error prone implementations, that were improved in order for the tests to succeed under various input permutations.

The purpose of the implemented tests was to evaluate the behavior of the API endpoints of the LTI Tool Client and Provider and provide a solid foundation for this project to facilitate extensibility and adaptability to new technologies while guaranteeing a method of testing and identifying regressions. The Javascript code of the user interface that is part of the LTI Tool Client was not tested, since that implementation is a simple consumer of the HTTP API, that was developed to provide better insights regarding the implemented functionality for the reader.

The routes of Table 4.1 have been tested using integration and unit tests. Table 4.3 shows the type of tests that were developed for each route, or middleware.

Table 4.3: Types of implemented tests per Endpoint

| Route | Unit tests | Integration tests |
|---|:---:|:---:|
| **POST** `/admin/login` | ✓ | ✓ |
| **GET** `/admin/logout` | ✓ | ✓ |
| **POST** `/admin/containers/run/:id` | ✗ | ✓ |
| **DELETE** `/admin/containers/kill/:id` | ✗ | ✓ |
| **POST** `/admin/containers/commit/:id` | ✗ | ✓ |
| **GET** `/admin/images` | ✓ | ✓ |
| **GET** `/admin/images/history/:id` | ✓ | ✓ |
| **DELETE** `/admin/images/delete/:id` | ✓ | ✓ |
| **POST** `/lti/launch/:id` | ✓ | ✗ |
| Session authorization middleware | ✓ | ✓ |
| LTI OAuth middleware | ✓ | ✗ |

In addition to the unit tests for the HTTP routes, unit tests were written to test the functionality of other Go packages that were developed as dependencies of this project, such as the `portmapper` package that implements the algorithm introduced in Section 4.1.6, for repositories of the `interface` layer, and finally the `api` package that contains the model definitions along with several helper functions for standardizing API responses and error handling. The endpoints prefixed with `/admin/containers/` were tested using integration tests only. This route group depends solely on docker containers being in a particular state, in order to test some specific action, therefore mocking the docker daemon would not provide more insights regarding the correctness of such software. In addition, all software dependencies of these routes, such as the session Redis repository, or the port binding software, has been already tested using unit tests. The unit tests for the rest of the API routes, were performed using mocked software dependencies. The LTI launch route was tested using unit tests, to check the functional correctness of OAuth middleware along with the route and the session management logic.

Table 4.4 presents the average execution time for each API route, given a valid request and a successful response. Each test executed for 100 times, and the logged time that was introduced in Section 4.4.3 for each route was logged to a file. Later, the average execution time was computed. Then tests run sequentially for each endpoint, using the `go test` command line for executing Ginkgo specifications.

Table 4.4: Average execution time for each endpoint

| Route | Mean execution time (ms) |
|---|---|
| **POST** /admin/login | 75.4567226 |
| **GET** /admin/logout | 0.21936022 |
| **POST** /admin/containers/run/:id | 598.56898125 |
| **DELETE** /admin/containers/kill/:id | 426.9313529 |
| **POST** /admin/containers/commit/:id | 542.58344475 |
| **GET** /admin/images | 11.56397094 |
| **GET** /admin/images/history/:id | 1.38308949 |
| **DELETE** /admin/images/delete/:id | 22.59314895 |

The most time consuming operations were the ones that required the creation of docker resources such as containers and images. The average execution time for a container run request is `598.56898125`ms which involves creating a container from a given image, and then starting the container.

While unit and integration tests are available in Travis builds, the performance tests were executed in physical machine. CPU and memory resources of Travis are often shared by multiple virtual machines, thus, measuring execution time of such software would depend on the system load at any given moment. The configuration settings of the machine used to perform the benchmarks are listed in detail in Appendix A. The modified code along with the time reports (stored in .csv files) for each specification can be found in the branch `report/benchmarks` of the GitHub repository [97] of this project.

# Chapter 5

# Conclusions and Future Work

LMSs are designed to improve learning, teaching and administrative tasks in higher education. Among their most important features is the integration of external applications that provides personalized domain specific e-learning. Student understanding of Computer Science (CS) domains such as computer networks involve in their curriculum hands-on experience via exercise material and laboratory practice. Such practice is usually performed within a traditional physical classroom and computer labs, and little progress has been made to offer similar learning experience within the context of a virtual classroom and e-learning. This thesis project investigated the integration of on-demand virtual laboratory environments for Internetworking e-learning with Canvas LMS leveraging the capabilities of Docker container virtualization. The outcome of this work is a software artifact that provides a method for instructors to easily build, manage and integrate virtual laboratory environments that are available to students as exercise material through an LMS.

## 5.1 Conclusions

One of the main points of this thesis is that a student or instructor can dynamically instantiate virtual exercise environments within a reasonable upper bounded time. The basic benchmarks that were presented in Section 4.4.4 indicated that accessing such environments can be achieved within less than a second, while their integration with the LMS improved their accessibility, comparing to traditional labs, where students and instructors are required to be physically present. Moreover, the preparation and configuration of the exercises can be performed as easily and within the same upper bounded time with accessing the environments. With traditional computer laboratories, institutions have a high cost for setting up and maintaining their own infrastructure, that includes human and server resources. Often such setups, have high utilization during pre-defined periods of the academic calendar, while their overall usage throughout a year is pretty low. Leveraging cloud technologies, one can argue that

such cost can be reduced and at the same time utilize the clouds' burst capabilities to serve the high demand during periods of the academic year while empowering distant learning.

The design science research methodology that was followed during this work, resulted to in-depth analysis of related work such as INGInious and the LTI specification, that concluded the initial system architecture. In addition, using existing software, allowed for a proof of concept implementation that fulfilled the initial design goals, while setting strong foundations for a future implementation of a scalable virtual laboratory that empowers Internetworking e-learning.

The LTI specification is well documented and simple to understand, since it is based on the widely known implementation of the OAuth authentication protocol. Although this work was limited to the LTI specification, the underlying work with the API and the Docker containers, allows for implementing several interoperability protocols, and expose them as different API endpoints, that can offer integration capabilities with several LMSs.

Choosing the Go programing language for the source code implementation of this project proved to be easier than expected, since the language is simple to learn and understand and widely adopted by the open source community that offers high quality documentation and examples for performing various programming tasks. In addition, the built-in support of the language for testing and benchmarking methodologies, improves the quality of source code, and drives developers to write simple, extensible and readable code.

The chosen method for accessing docker containers was a web based emulator of an SSH connection to a remote server. Although such choice proved to be particularly useful for testing and presenting the implementation of an LTI Launch case, it had several drawbacks. The host system required to expose a series of ports in order to allow a predefined number of containers to run and forward network packets from the host system to the docker daemon and the API, and at the same time exposed such ports to the user. The jQuery framework used to develop the web interface of the LTI Tool Client faced several problems when requesting resources from multiple hosts such as the API running on port 8080 and the containers running on a range of ports of the same server. A production-ready implementation of such system would run the API server in a different environment than the Docker server, and potentially use some sort of web proxy to route traffic between those servers without the user being aware of the underlying network configuration.

Finally, the use of technologies such as GitHub, Travis CI and CodeCov, contributed positively to the sustainability of this work, by enabling the reader to inspect the source code, the configuration settings of the development environment in a remote cloud infrastructure, and the evaluation methods used to test the software artifact. In addition, other software dependencies such as the docker images and the Canvas LMS setup has been version controlled in Docker Hub and GitHub repositories respectively.

## 5.2 Future work

This section introduces various topics that should be investigated as part of the future work related to this project. Section 5.2.1 addresses different approaches to scaling the LTI Tool Client and its docker dependencies. Section 5.2.2 lists alternative implementations for web based shell emulators. Section 5.2.3 discusses future functionality of the LTI Tool Client. Section 5.2.4 lists approaches to evaluate the performance and implementation of the "Periodic Checker" module. Section 5.2.5 documents functionality that should be implemented to improve the performance, stability, and usability of the TP. Finally, Section 5.2.6 discusses ideas for supporting automatic evaluation of assignments. During the development process of this project various issues were discovered.

### 5.2.1 Scalability

The architectural design of the implementation presented in Chapter 4 has limited scalability. The Docker daemon and the TP are running in the same virtual machine on one physical computer system, hence they are bound by the CPU and memory resources that the underlying physical machine provides, hence the containers aggregate resource consumption is limited. In addition, this setup has limitations on the number of ports the Docker daemon can use to bridge network connections between the containers and the host system. A lot of work has been done in deploying scalable clusters of container runtime environments. Docker Swarm [104] has clustering capabilities for turning groups of Docker Engines into a single, virtual Docker Engine. Swarm treats each Docker Engine as a node of a decentralized distributed system, and offers a series of features such as load balancing and methods for scaling applications running in a cluster. This functionality is available via the Docker Swarm API[105] that is designed to be (mostly) compatible with the Docker Remote API*.

### 5.2.2 Web based shell emulators

Shell In A Box was chosen as the web terminal emulator, but there are alternative implementations that have not being investigated as this was outside the scope of this project. In addition, the base container image used by the Tool Client was not evaluated. The configuration of the shellinabox software package has more capabilities than supported in the docker image `sspreitzer/shellinabox`. Some of these additional features that may prove useful in this project are: predefined TLS certificates for the Shell in a Box server, specifying Linux user groups, usernames, disabling or enabling *sudo* access for users of a particular container image, and customization of the CSS of the web emulator.

---

*Some API endpoints of Docker Remote API have not yet been implemented in the Docker Swarm API. These missing endpoints are documented in the official documentation page [105].

Section 2.8 presented three configuration parameters of the `sspreitzer/shellinabox` container image, that were used in this project. These parameters were `SIAB_USER` that defines the username to be used when accessing the container via an SSH session, `SIAB_PASSWORD` that defines the user's password, and `SIAB_SUDO` that provides the user with *sudo* access. During the implementation of the "Commit Container" functionality for the Tool Client, it was decided that all users would have the username "guest", but different passwords.

When a container of the chosen image is created (i.e. for the admin user of the Tool Client), a default user group and identifier is chosen for the given user. If a new image is created using this container, this user group and identifier are reserved for the admin user. As a consequence, when running a container of this newly created image, the default settings will collide with the ones committed by the admin, hence the container fails to start the Shell in a Box web server. Discarding the `SIAB_USER` parameter from the creation process of a container, resulted in the use of the default "guest" user and solved the problem temporarily, but created additional security issues such as, any user knowing the port number and the password of another user can access their running shell. To overcome this issue, a new method should be implemented using the entrypoint.sh[†] and the Dockerfile[‡], that deletes the user group, the user, etc., used by the admin when an image is created using a running container.

This project assumed that a web based emulator would be a suitable method for accessing a laboratory environment. As part of future work, an alternative implementation should be investigated, that instead of returning an emulator shell, directly provides configuration settings, such as an ssh key for download, so that the user can ssh directly into the container from her own terminal. This will probably be more useful than using a shell running in a browser window, as users will not rely on their browser communicating with the emulator, but instead will rely directly on their local ssh agent.

### 5.2.3 Tool Client user interface design

The design of the Tool Client was not based on user research, i.e., how the user expects to access an emulator shell and what views are desired by an admin to easily manage container images. In addition, the implemented functionlity of the Tool Client is very limited. An instructor will want to have a way of knowing which images are created and which assignments these container images are associated with. A user-oriented approach should be used, to identify the most important functionality for the admin user and students. In addition to defining the required functionality, the method of presenting this functionality should be

---

[†]The file entrypoint.sh of the {sspreitzer/shellinabox} container image is a bash script that initializes parameters required by the web server to start.

[‡]The Dockerfile is text configuration file that contains all the commands a user could pass on a command line to assemble an image.

investigated. This could use methodologies of "User Experience Design"[106] such as wireframes, prototypes, and user stories. Additionally, the system should be evaluated using real users, in order to identify usability issues and limitations, while exploring alternatives.

### 5.2.4 Evaluation of the Periodic Checker module

Section 4.1.6 introduced the functionality of the `PeriodicChecker` module. The algorithm explained in that section is based on several assumptions, such as the availability of the Redis session storage and that a periodic check will complete in sufficient time to avoid timeouts of HTTP user requests that are handled by mutex locked goroutines. For example, during a scheduled check, the module identifies that a container port is no longer used and it marks its value in the `PortResources` map as `false`. Right after that a request to Redis session storage is performed to remove any keys associated with that port. The request to Redis is performed synchronously and the function waits for a response, hence every goroutine that is trying to access the `PortResources` map, will be blocked until Redis replies. A request to run a container might time out due to this waiting. As part of future work, the system should be benchmarked and tested against such scenarios, to decide whether the Redis session storage is appropriate for storing running container configurations.

In addition, Go provides an additional mechanism for accessing memory resources concurrently. This mechanism is called a *channel*, and it allows a goroutine to send values to another goroutine. The functionality of channels should be investigated, and compared against the implementation of mutex locks, to identify whether such an implementation would be beneficial for performing atomic operations on the `PortResources` map.

### 5.2.5 Desired Features

During the implementation of the Tool Client, various use cases were explored and these inspired the features described in this section. The desired features are:

- The system should evaluate if an image is functional after it is committed. A functional image is defined as a container image that can successfully launch the web SSH emulator process. A process should be implemented that tests a newly created image against such criteria, and if an image fails to pass, then the admin user of the Tool Client should be notified and the image should be flagged as problematic.

- The RDBMS schema should be extended to include information that associates a container image with an assignment along with additional information provided by the admin user of the Tool client. This information should be visible in the Tool Client.

- The implementation of the session storage mechanism does not allow a user to run multiple container images at the same time. This is a limitation, as a

99

student might want to complete multiple assignments at the same time and the admin user might wish to launch multiple container images to verify or copy configurations.

- The Tool Client should support a page that presents the admin user with a page providing additional useful information, such as which containers are running at the time the page is requested and which users are associated with each container. In addition, such a page could provide analytics about the previous usage of the containers, such as the duration of each currently running container session, average and mean execution times per image (or assignment), etc.

- The user interface of the LTI Launch and the Tool Client performs synchronous requests to the server to implement requests to running a container (i.e., HTTP POST requests to the `/admin/containers/run/:id` and HTTP POST requests to `/lti/launch/:id`). The interface assumes that after a successful response from the server, the requested container will continue running. However, if a container crashes, the user is not informed. An implementation that asynchronously checks (for example, by doing heartbeat monitoring) whether the requested container is still running along with methods to present failure results to the user should be investigated.

### 5.2.6 Assignment evaluation

Part of the initial idea for this project, was to design a system that supports automatic evaluation of assignments and reporting of analytics that will assist in the learning process for both students and instructors. Section 2.9.3 explained how INGInious supports automatic evaluation of coding assignments with unit testing. Unfortunately, Internetworking assignments have different requirements than a coding assignment. As part of future work, such requirements should be investigated to conclude if similar unit-testing approaches can be used to evaluate Internetworking assignments.

In addition to the evaluation, an instructor is often interested in how much time a student takes to complete an assignment. Several time-tracking approaches could be used to extract such analytics for an instructor.

# References

[1] Computer System Engineering - M.I.T. Department of EECS. Understanding TCP using tcpdump. `http://web.mit.edu/6.033/www/assignments/handson-tcp.html`. [Online; accessed 2017-01-02].

[2] William R. Watson and Sunnie Lee Watson. An argument for clarity: what are learning management systems, what are they not, and what should they become? *TechTrends*, 51(2):28–34, 2007.

[3] Stefan Boesen, Richard Weiss, James Sullivan, Michael E. Locasto, Jens Mache, and Erik Nilsen. EDURange: Meeting the Pedagogical Challenges of Student Participation in Cybertraining Environments. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, San Diego, CA, August 2014. USENIX Association.

[4] Ricardo Nabhen and Carlos" Maziero. *Education for the $21^{st}$ Century — Impact of ICT and Digital Resources: IFIP 19th World Computer Congress, TC-3, Education, August 21–24, 2006, Santiago, Chile*, chapter Some Experiences in Using Virtual Machines for Teaching Computer Networks, pages 93–104. Springer US, Boston, MA, 2006.

[5] Introducing hands-on experience to a massive open online course on openhpi.

[6] Instructure, Inc. Canvas Learning Management System. `https://www.canvaslms.com/`. [Online; accessed 2016-02-21].

[7] Daniela Fonte, Daniela da Cruz, Alda Lopes GanÃ§arski, and Pedro Rangel Henriques. A flexible dynamic system for automatic grading of programming exercises. In *OASIcs-OpenAccess Series in Informatics*, volume 29. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[8] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a MOOC using the INGInious platform.

[9] Ricardo Queirós and José Paulo Leal. Programming Exercises Evaluation Systems - An Interoperability Survey. In *CSEDU (1)*, pages 83–90, 2012.

[10] Alan Hevner and Samir Chatterjee. Design Science Research in Information Systems. In *Design Research in Information Systems*, volume 22, pages 9–22. Springer US, Boston, MA, 2010.

[11] Vijay K. Vaishnavi and William Kuechler, Jr. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach Publications, Boston, MA, USA, 1st edition, 2007.

[12] Alan R. Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

[13] C. Alario and S. Wilson. Comaprison of The Main Alternatives To The Integration of External Tools in Different Platforms. In *ICERI2010 Proceedings*, 3rd International Conference of Education, Research and Innovation, pages 3466–3476. IATED, 15-17 November, 2010 2010.

[14] Open edX as an LTI Tool Provider. `https://open.edx.org/blog/open-edx-lti-tool-provider`. [Online; accessed 2016-02-28].

[15] Md. Iqbal Hossain and Md. Iqbal Hossain. Dynamic scaling of a web-based application in a Cloud Architecture. Master's thesis, KTH, Radio Systems Laboratory (RS Lab), 2014.

[16] Ryann K Ellis. Field Guide to Learning Management Systems, 2009.

[17] José Paulo Leal and Ricardo Queirós. A comparative study on LMS interoperability. *Higher Education Institutions and Learning Management Systems: Adoption and Standardization*, page 142, 2011.

[18] Wynne Harlen and Mary James. Assessment and Learning: differences and relationships between formative and summative assessment. *Assessment in Education: Principles, Policy & Practice*, 4(3):365–379, November 1997.

[19] Janne Malfroy Kevin Ashford-Rowe. E-Learning Benchmark Report: Learning Management System (LMS) usage. `http://www.uws.edu.au/__data/assets/pdf_file/0007/452077/Griffith_UWS_Elearning_Benchmark_Report.pdf`, 2009.

[20] ISO. Information Technology Vocabulary. ISO 2121317 - 2382:2015, International Organization for Standardization, 2015.

[21] IMS GLOBAL Learning Consortium. Learning Tools Interoperability ®(LTI®). `http://www.imsglobal.org/activity/learning-tools-interoperability`. [Online; accessed 2016-02-23].

[22] Ricardo Queirós, José Paulo Leal, and José Paiva. Integrating rich learning applications in LMS. In *State-of-the-Art and Future Directions of Smart Learning*.

REFERENCES

[23] IMS Learning Information Services. `https://www.imsglobal.org/lis/`. [Online; accessed 2016-02-28].

[24] Ruby Sinatra - official documentation page. `http://www.sinatrarb.com/documentation.html`. [Online; accessed 2016-07-17].

[25] Alan Harris and Konstantin Haase. *Sinatra: Up and Running.* O'Reilly Media, Inc., 1st edition, 2011.

[26] LTI Outcome Service Example using Canvas LMS. `https://github.com/instructure/lti_example`. [Online; accessed 2016-04-23].

[27] IMS Global Learning Consortium including the IMS Logos, Learning Tools Interoperability® (LTI®). IMS Global General Web Services. `https://www.imsglobal.org/gws/index.html`. [Online; accessed 2016-07-27].

[28] IMS Global Learning Consortium including the IMS Logos, Learning Tools Interoperability® (LTI®). IMS Global Learning Tools Interoperability™ Implementation Guide. `https://www.imsglobal.org/specs/ltiv1p1/implementation-guide`. [Online; accessed 2016-07-27].

[29] Ed. E. Hammer-Lahav. The OAuth 1.0 protocol, April 2010.

[30] OpenSSL Software Foundation. OpenSSL cryptography and SSL/TLS toolkit. `https://www.openssl.org/`. [Online; accessed 2016-08-07].

[31] Marcus Redivo. Creating and using SSL certificates. `http://www.eclectica.ca/howto/ssl-cert-howto.php`. [Online; accessed 2016-08-07].

[32] OpenSSL Software Foundation. OpenSSL - official documentation of command `req`. `https://www.openssl.org/docs/manmaster/apps/req.html`. [Online; accessed 2016-08-07].

[33] Phil Dibowitz. Openssl.conf walkthru. `https://www.phildev.net/ssl/opensslconf.html`. [Online; accessed 2016-08-07].

[34] An open LTI app collection. https://www.eduappcenter.com/. [Online; accessed 2016-07-11].

[35] Instructure Inc. Instructure. `https://www.instructure.com/`. [Online; accessed 2016-07-17].

[36] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to Support PaaS. pages 610–614. IEEE, March 2014.

[37] Linux Containers - LXC. `https://linuxcontainers.org/lxc/introduction/`. [Online; accessed 2016-02-28].

[38] Linux Programmer's Manual, overview of Linux namespaces. `http://man7.org/linux/man-pages/man7/namespaces.7.html`. [Online; accessed 2016-02-28].

[39] Rami Rosen. Linux containers and the future cloud. *Linux J*, 240, 2014.

[40] Docker Inc. Docker. `https://www.docker.com/`. [Online; accessed 2016-02-28].

[41] Docker Inc. Libcontainer implementation. `https://github.com/opencontainers/runc/tree/master/libcontainer`. [Online; accessed 2016-11-20].

[42] LXC container driver. `https://libvirt.org/drvlxc.html`. [Online; accessed 2016-11-20].

[43] systemd-nspawn. `https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html`. [Online; accessed 2016-11-20].

[44] The Linux Foundation®. Open Container Initiative. `https://www.opencontainers.org/about`. [Online; accessed 2016-11-20].

[45] Docker Inc. Docker Command Line Reference. `https://docs.docker.com/engine/reference/commandline`. [Online; accessed 2016-11-20].

[46] Docker Remote API. `https://docs.docker.com/engine/reference/api/docker_remote_api`. [Online; accessed 2016-11-20].

[47] Michael Kerrisk. Unix pseudoterminal interface. `http://man7.org/linux/man-pages/man7/pty.7.html`, 2005. [Online; accessed 2016-11-20].

[48] Web based SSH. `https://en.wikipedia.org/wiki/Web-based_SSH`. [Online; accessed 2016-11-19].

[49] Dan McDougall. GateOne. `https://github.com/liftoff/GateOne`. [Online; accessed 2016-11-20].

[50] Markus Gutschk. shellinabox.

[51] Digital Equipment Corporation. *VT100 Series Technical Manual*, 1979.

[52] Holdener, III, Anthony T. *Ajax: The Definitive Guide*. O'Reilly, first edition, 2008.

[53] Sascha Spreitzer. shellinabox for docker. `https://github.com/sspreitzer/docker-shellinabox`. [Online; accessed 2016-11-20].

[54] The Linux Foundation. The Linux foundation wiki - bridge. `https://wiki.linuxfoundation.org/networking/bridge`. [Online; accessed 2016-12-18].

REFERENCES

[55] Docker Inc. Docker documentation - customize the docker0 bridge. `https://docs.docker.com/engine/userguide/networking/default_network/custom-docker0/`. [Online; accessed 2016-12-18].

[56] The Evergreen State College Olympia, Washington. EDURange: A Cybersecurity Competition Platform to Enhance Undergraduate Security Analysis Skills. `http://blogs.evergreen.edu/edurange/`. [Online; accessed 2016-02-28].

[57] Amazon Elastic Compute Cloud - Amazon EC2. `https://aws.amazon.com/ec2/`. [Online; accessed 2016-02-28].

[58] EDURange Github project. 2014. [Online; accessed 2016-02-28].

[59] Carlos Alario-Hoyos, Miguel L. Bote-Lorenzo, Eduardo Gómez-Sánchez, Juan I. Asensio-Pérez, Guillermo Vega-Gorgojo, and Adolfo Ruiz-Calleja. GLUE!: An architecture for the integration of external tools in Virtual Learning Environments. *Computers & Education*, 60(1):122–137, 2013.

[60] INGinious by Université Catholique de Louvain. `http://inginious.org/`. [Online; accessed 2016-02-28].

[61] Github repository of INGInious. `https://github.com/UCL-INGI/INGInious`. [Online; accessed 2016-02-28].

[62] Technical documentation of INGInious. `http://inginious.readthedocs.org`. [Online; accessed 2016-02-28].

[63] Teacher documentation of INGInious. `http://inginious.readthedocs.io/en/latest/teacher_documentation.html`. [Online; accessed 2016-02-28].

[64] Vijay K. Vaishnavi and William Kuechler, Jr. Design Science Research in Information Systems. January.

[65] Ruby on Rails - official web page. `http://rubyonrails.org/`. [Online; accessed 2016-07-17].

[66] Instructure, Inc. Canvas LMS Istallation Quick Start Wiki Page. `https://github.com/instructure/canvas-lms/wiki/Quick-Start`. [Online; accessed 2016-11-07].

[67] HashiCorp. Vagrant. `https://www.vagrantup.com/`. [Online; accessed 2016-08-07].

[68] Oracle. VirtualBox. `https://www.virtualbox.org/`. [Online; accessed 2016-08-07].

[69] Andreas Kokkalis. Canvas LMS installation using Vagrant. `https://github.com/andreas-kokkalis/canvas_lms_vagrant`. [Online; accessed 2016-08-07].

[70] The Go Programming Language. `https://golang.org/`. [Online; accessed 2016-08-07].

[71] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[72] Go package net/http. `https://golang.org/pkg/net/http/`. [Online; accessed 2016-08-07].

[73] Julien Schmidt. HttpRouter - a trie based high performance HTTP request router. `https://github.com/julienschmidt/httprouter`. [Online; accessed 2016-12-07].

[74] Docker Inc. Go implementation of the Docker Remote API library. `https://godoc.org/github.com/docker/docker/client`. [Online; accessed 2016-08-07].

[75] Go package context. `https://golang.org/pkg/context/`. [Online; accessed 2016-12-07].

[76] The Go Programming Language Specification. `https://golang.org/ref/spec#The_zero_value`. [Online; version 2016-05-31].

[77] Docker Inc. Docker Checkpoint and Restore. `https://github.com/docker/docker/blob/master/experimental/checkpoint-restore.md`. [Online; accessed 2016-12-18].

[78] Redis. `https://redis.io/`. [Online; accessed 2016-12-18].

[79] Type-safe Redis client for Golang. `https://github.com/go-redis/redis`. [Online; accessed 2016-12-18].

[80] PostgreSQL - open source object relational database system. `https://www.postgresql.org/`. [Online; accessed 2016-12-18].

[81] Go package database/sql. `https://golang.org/pkg/database/sql/`. [Online; accessed 2016-12-18].

[82] Pure Go Postgres driver for database/sql. `https://github.com/lib/pq`. [Online; accessed 2016-12-18].

[83] The jQuery Foundation. ajax function of jquery for performing asynchronous http (ajax) requests. `https://api.jquery.com/jquery.ajax/`. [Online; accessed 2016-12-18].

REFERENCES

[84] J.Reschke R. Fielding. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, June 2014.

[85] tcpdump - command line packet analyzer. `http://www.tcpdump.org/`. [Online; accessed 2016-12-18].

[86] Jordi Collell. Golang LTI - Go Tools for working with the LTI specification. `https://github.com/jordic/lti`. [Online; accessed 2016-12-18].

[87] Software Testing Levels. `http://softwaretestingfundamentals.com/software-testing-levels/`. [Online; accessed 2017-12-29].

[88] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017.

[89] Manuel Kiessling. Applying the Clean Architecture to Go applications. `http://manuel.kiessling.net/2012/09/28/applying-the-clean-architecture-to-go-applications/`. [Online; accessed 2017-12-29].

[90] Go package testing. `https://golang.org/pkg/testing/`. [Online; accessed 2017-12-28].

[91] Rob Pike. The Go Blog, The cover story. `https://blog.golang.org/cover`. [Online; accessed 2017-12-28].

[92] IBM Knowledge Center. Source code instrumentation overview. `https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html`. [Online; accessed 2017-12-29].

[93] *gcov - A Test Coverage Program.* [Online; accessed 2017-12-29].

[94] ®Travis CI, GmbH. Travis CI. `https://about.travis-ci.com/`. [Online; accessed 2017-12-29].

[95] John Ferguson Smart. *Jenkins: The Definitive Guide.* O'Reilly Media, Inc., 2011.

[96] Travis builds of dock server github project. `https://travis-ci.org/andreas-kokkalis/dock_server/`. [Online; accessed 2017-12-29].

[97] Andreas Kokkalis. Github project of On-demand virtual laboratory environments for Internetworking e-learning: A first step using docker containers. `https://github.com/andreas-kokkalis/dock_server`. [Online; accessed 2017-12-29].

[98] Codecov. Home page of Code Coverage tool. `https://codecov.io/`. [Online; accessed 2017-12-29].

[99] Source code statement coverage reports of the dock server GitHub project. `https://codecov.io/gh/andreas-kokkalis/dock\_server`. [Online; accessed 2018-01-03].

[100] Onsi Fakhouri. A Golang Behavior Driven Design Testing Framework. `https://onsi.github.io/ginkgo/`. [Online; accessed 2018-01-03].

[101] J.F. Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle.* Manning Publications Company, 2014.

[102] Inc. Bazaarvoice. JSON to JSON diff tool. `https://github.com/bazaarvoice/json-regex-difftool`.

[103] Apache license, version 2.0.

[104] Docker Inc. Docker swarm. `https://www.docker.com/products/docker-swarm`. [Online; accessed 2017-01-02].

[105] Docker Inc. Docker swarm api. `https://docs.docker.com/v1.9/swarm/api/swarm-api/`. [Online; accessed 2017-01-02].

[106] Sari Kujala, Virpi Roto, Kaisa Väänänen-Vainio-Mattila, Evangelos Karapanos, and Arto Sinnelä. UX Curve: A Method for Evaluating Long-term User Experience. *Interact. Comput.*, 23(5):473–483, September 2011.

# Appendix A

# Development and testing setup

The development and testing of this project were performed using a laptop machine with the following setup:

- **Operating system**: Ubuntu Linux 16.04.3 Long Term Support (LTS).
- **CPU**: Intel® Core™ i7-7500U CPU  2.70GHz x 4
- **Memory**: 16GB
- **Go version**: go1.9.2 linux/amd64

The Docker engine installation had the following settings as extracted by executing the command `docker version`:

- **Version**: 17.12.0 Community Edition
- **API version**: 1.35 (minimum version 1.12)
- **Go version**: go1.9.2
- **Git commit**: c97c6d6
- **Built**: Wed Dec 27 20:09:53 2017
- **OS/Arch**: linux/amd64
- **Experimental mode**: enabled

The docker containers used as dependencies for Redis and PostgreSQL were:

- **Redis**: `redis:4.0` or `redis:latest`
- **PostgreSQL**: `postgres:9.6.1`

The containers used during implementation and testing can be found under the Docker Hub url `https://hub.docker.com/r/andreaskokkalis/dc/tags/`.