Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Master's thesis

Application for Creating, Taking and Evaluating Written Exams

Bc. Adam Havel

Supervisor: doc. Ing. Tomáš Vitvar, Ph.D.

6th May 2014

Acknowledgements

Declaration

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

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

In Prague on 6th May 2014 . . . .

Citation of this thesis

Havel, Adam. *Application for Creating, Taking and Evaluating Written Exams*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstrakt

Klíčová slova   Replace with comma-separated list of keywords in Czech.

Abstract

Keywords   Replace with comma-separated list of keywords in English.

Contents

List of Figures

xiii

Introduction

Realization

1.1 Architecture

The application structure is based on a concept that is usually referred to as the MEAN stack—a name that reflects the various technologies used to create web applications that operate on both the server and client side. The core of the application is powered by *Node.js* [1]—a platform built on *Google V8* JavaScript runtime that allows building fast and scalable network applications. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Express [2] is a *Node.js* web application framework which provides a higher-level abstraction of the native *Node.js* request-handling API and it is used primarily for its routing capabilities. *Express* automatically parses requests sent to the server and prepares appropriate responses. These are usually constructed by means of middleware which allows us to easily share functionality—such as an authentication layer—among different routes.

MongoDB [3] is a document-oriented database system, classified as a NoSQL database which means the traditional table-based relational database structure is eschewed in favor of JSON documents with dynamic schemas. This in return makes the integration of data in certain types of applications easier and faster. While not perfect, *MongoDB* is currently the most popular NoSQL solution.

Last but not least, *AngularJS* [4] is a Google-maintained web application framework that is used in the client-side part of the application in order to augment it with model–view–controller capability. The framework adapts and extends traditional HTML to better serve dynamic content through two-way data-binding that allows for the automatic synchronization of models and views. As a result, *AngularJS* de-emphasizes DOM manipulation and encourages loose coupling between presentation, data, and logic components. Furthermore, by using dependency injection, it brings traditional server-side services, such as view-dependent controllers, to client-side web applications. Consequently, much of the burden on the backend is reduced, leading to much lighter web applications.
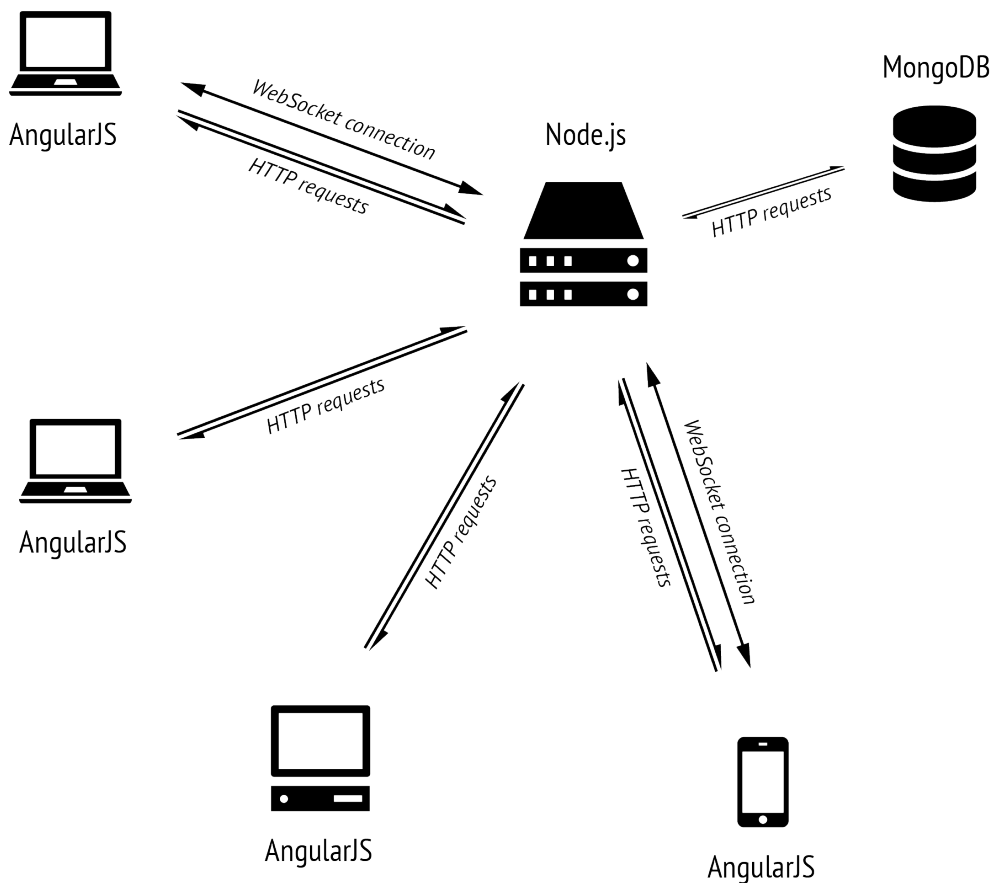

Figure 1.1: Application architecture

The common thread to all these distinct technologies is JavaScript which they all employ in some way. The advantages that arise from using a uniform language throughout the entire application are considerable. It allows us to store JSON-like documents in the *MongoDB* database, to write JSON queries on the *Node.js*-based server, and to seamlessly pass JSON documents to the *AngularJs* frontend. When the objects stored in the database are essentially identical to the objects the client application sees, development and testing become a lot easier. Even better, using the same syntax and objects the whole way through greatly reduces the barrier to understand the code.

1.2 Server

The server part of the application is written entirely in *Node.js* and its main purpose is to provide access to the database by means of API, to handle real-time communication between users using the WebSocket protocol, and to serve static files containing the client application.

The static files are physically located in the `public` folder. However, when served, they must be accessed relative to the application root URL. Any request that does not point at a static resource such as an image or a style sheet is redirected to the file `public/index.html` which holds the *AngularJS* application—the routing is then handled by the client. An exception to this rule is the server's API which is located at `/api`. Such requests are further processed by the server.

The server-side code is logically divided into three categories. The first one stands for models which are the data models used both in the database and in the application. The second category are routes that are used to define endpoints for resources. Their behavior is then provided by controllers which represent the last group.

1.2.1 Database

Using *MongoDB* as the database allows us to define stored objects in a [5]

1.2.1.1 Data models

The key decision when designing data models for *MongoDB* revolves around the structure of documents and their relationships between each other. There are essentially two ways how to approach these relationships: references and embedded documents.

References store the relationships between data by including links or references from one document to another. The application can resolve these references later and access the related data. Using references properly results in the so-called normalized data models. Embedding documents, on the other hand, means storing related data in a single document structure. *MongoDB* documents make it possible to embed document structures as sub-documents in a field or array within a document. These denormalized data models allow the application to retrieve and manipulate related data in a single database operation. In the end, each of these approaches has its merit and there is no reason not to employ a combination of both if it proves useful or efficient.

The application contains three data models: blueprints, exams and users. A blueprint represent what in real life would be the original copy of a written test. The figure 1.2 displays the *Mongoose* schema that defines blueprints in the database. They are identified by a combination of a subject, a date and a language. The schema also reveals that blueprints hold an array of sections which take the form of embedded documents. Each of these sections then contains yet another array of questions. The reason why we specifically have to define the schema for questions and not for the other sub-documents will be explained shortly. The `Schema.Types.Mixed` type in the schema points at the fact that the structure of a content is not known beforehand, and can take many forms depending on the value of the `datatype` property.

Exams stand for students' test, they are therefore closely connected to blueprints. One approach would be to embed the original blueprint into every related exam but given the fact that blueprints usually hold more data than exams, the overhead would be too great. It is therefore more sensible to use references—in this case defined as the `_blueprint` property which links the blueprint in question by means of a `Schema.Types.ObjectId` identifier that contains the `_id` property of the linked document.

6

```
var BlueprintSchema = new Schema({
  subject: String,
  date: Date,
  lang: String,
  lede: String,
  sections: [
    {
      name: String,
      lede: String,
      points: Number,
      questions: [QuestionSchema]
    }
  ]
});

BlueprintSchema.index({
  'subject': 1,
  'date': -1,
  'lang': 1
});

var QuestionSchema = new Schema({
  name: String,
  points: Number,
  body: [
    {
      datatype: String,
      lang: String,
      content: Schema.Types.Mixed
    }
  ],
  answer: [
    {
      datatype: String,
      lang: String,
      content: Schema.Types.Mixed,
      solution: Schema.Types.Mixed
    }
  ]
});
```

Figure 1.2: Blueprint schema

Looking at `ExamSchema` in the figure 1.3 reveals that apart from holding a student's fingerprint as an embedded document, an exam also contains the array `answers`. While confusing at first, each of these answers actually represents the same structure as the items in blueprints' `questions`. Only the body property is stripped and the `asnwer` is renamed in order to take its place. We must also reference the original question using the `_question` property, since the answers no longer fall withing sections—as they do in blueprints. This helps when linking the two together and means that we do not have to rely on the items' order in the arrays. It is also the reason why we have to define `QuestionSchema`—otherwise, we would not be able to reference questions in exams. When set in this manner, the server can replace the references with actual data by using the *Moongoose* command `populate()` whenever a client requests an exam.

Another point of interest is the property `evaluted` which allows the application to tell whether an exam has been evaluated or not—it contains either `null` or a sub-document with information about the evaluation itself such as its author.

The last distinct collection in the database consists of users. An entry is created whenever a user logs in the application for the first time. The associated data is then stored using the schema in the figure 1.4. Unlike the previous schemata, `UserSchema` overrides the native `_id` property—normally holding an `ObjectID` value—with a string holding the user's unique identifier, defined in the university information system *KOS*.

1.2.2 Internal communication

The communication between the client and the server can take place on two different levels. The first one depends on the server's public API that provides access to a set of resources. These are fetched by the client using asynchronous XHR-type requests. The other one is represented by a full-duplex WebSocket connection that allows real-time communication between users—the server serving as an intermediary.

```
var ExamSchema = new Schema({
  student: {
    name: String,
    id: String
  },
  _blueprint: {
    type: Schema.Types.ObjectId,
    ref: 'Blueprint'
  },
  evaluated: { type: Schema.Types.Mixed, default: null },
  subject: String,
  date: Date,
  lang: String,
  answers: [
    {
      _question: {
        type: Schema.Types.ObjectId,
        ref: 'Question'
      },
      body: [Schema.Types.Mixed],
      points: { type: Number, default: 0 },
    }
  ]
});

ExamSchema.index({
  'subject': 1,
  'date': -1,
  'lang': 1,
  'student.id': 1
});
```

Figure 1.3: Exam schema

```
var UserSchema = new Schema({
  _id: String,
  name: String,
  role: String,
  subjects: [String],
  lastLogin: Date
});
```

Figure 1.4: User schema

1.2.2.1 API

The server-side API essentially represents a doorway to the database, each of the available resource mirroring a collection in the store. Since it is not meant to be consumed by any other application—not in the current revision anyway—its purpose is limited to being the link between the client and the server. The API is therefore designed primarily to conform to this goal, and while based on the principles of REST, it cannot be considered RESTful [6]. For that to be true, the resources would have to be represented as hypermedia, which would in turn render the API self-discoverable. In the current form, the client must have knowledge of the API structure and available methods in advance.

Access to the API is forbidden unless authenticated, and depending on the resource and parameters provided, an additional authorization layer might be present. Some resources are available just to privileged users or only when certain conditions are met. An example of a resource—as defined by *Express*—can be seen in the figure 1.5, including the authorization middleware function. A resource's behavior can be specified by creating a sequence of such functions for each of the HTTP methods the resource allows. In order to be considered a middleware, these functions must possess the signature (`req`, `res`, `next`) where `req` represents the request that started the sequence, `res` can be used to set and send a response, and `next` points at the next function in line. If an error occurs, the current middleware can abort the sequence and send an error message via the `res` object. If not, the request is passed along to the next middleware by calling `next()`. In the end, the last function in the sequence should send some kind of response back to the client.

1.2.2.2 Websockets

The WebSocket protocol [7] is a web technology that provides a full-duplex communication channel over a single persistent TCP connection, without the overhead of HTTP headers and cookies. This is made possible by introducing a standardized way in which the server can push content to the client without being requested, and by allowing to pass messages back and forth

```
module.exports = function(app) {

  function isAuthorized(req, res, next) {
    if (req.params.subject && _.indexOf(req.user.subjects, req.
        params.subject) === -1) {
      return res.send(403, 'Access forbidden');
    }
    next();
  };

  app.route('/api/blueprints')
     .get(auth.isAuthenticated, isAuthorized, blueprints.query);

  app.route('/api/blueprint/:subject/:date/:lang')
     .get(auth.isAuthenticated, isAuthorized, blueprints.get)
     .post(auth.isAuthenticated, auth.isTeacher, isAuthorized,
        blueprints.create);

  app.param('subject', exams.validateSubject);
  app.param('date', exams.validateDate);
  app.param('lang', exams.validateLang);

};
```

Figure 1.5: Example of a resource definition

while keeping the connection open.

Socket.IO [8] is an event-driven library that is employed to further expand upon the basic functionality of WebSocket. Although it can be used simply as a wrapper around the protocol—to bring WebSocket-like functionality to older browsers by means of fallbacks—it also provides otherwise unavailable features such as broadcasting to multiple sockets or storing data associated with each client. The library has two parts: one that runs in the browser, and other that requires a *Node.js* server.

The protocol is primarily utilized for handling ongoing exams. The moment a user enters an exam, a new WebSocket connection is established. The user then identifies himself to the server which in turn stores the user's credentials alongside the socket that handles this particular communication. The socket has its own identifier and contains additional information about the connection such as the client's IP address. After that, the user is allowed to

join a room that holds the same name as the exam he is taking—room being a concept introduced by *Socket.IO*. There can be many rooms and each represents an ongoing exam which allows us to broadcast messages only to a relevant subset of users.

```javascript
socket.on('exam:start', function(data) {
  var exam = {
    id: data.examId,
    start: Date.now(),
    duration: data.duration,
    clock: setInterval(function() {
      exam.timeLeft = exam.duration - (Date.now() - exam.start);
      if (exam.timeLeft <= 0) {
        if (io.sockets.clients(exam.id).length) {
          io.sockets.in(exam.id).emit('exam:finished');
        } else {
          clearInterval(exam.clock);
          _.remove(exams, function(item) {
            return item.id === exam.id;
          });
        }
      }
    }, 1000)
  };
  exams.push(exam);
  io.sockets.in(data.examId).emit('exam:started', {
    start: exam.start,
    duration: exam.duration
  });
});
```

Figure 1.6: Example of a *Socket.IO* event

When a user reloads the page that holds the connection or closes the browser, the socket is lost. It is therefore necessary to check the user's credentials against the already stored users whenever a new socket is made. If a match is found, the server compares the socket that has been saved alongside the matching user with the socket handling the communication. If not equal, the old socket is replaced with the new one. That way, a one-to-one mapping between users and sockets is ensured.

During an ongoing exam, the server must handle communication between users present at the exam which entails processing incoming messages, and if

necessary, sending or broadcasting messages to a particular user or a subset of users—usually defined by their roles. An example of a message handler is shown in the figure 1.6. In this case the server receives a `exam:start` message from a user and in return starts an internal timer and notifies all users in the relevant room by sending a message with the value `exam:started`. When the timer runs out, a `exam:finished` prompt is periodically sent until no user remains in the room. When a user receives this message, he leaves the exam and sends a confirmation to the server which responds by finally removing the user from the list.

1.2.3 Integration of external resources

The application uses two external resource: the faculty LDAP server running at `ldap.fit.cvut.cz` and the *KOSapi* service which provides a REST API over the information system *KOS*.

1.2.3.1 LDAP authentication

The LDAP server is utilized as a means for secure authentication of users. To authenticate, a user enters his credentials in the application login form on the client side. These are then passed to the application server which in turns asks the LDAP server. This part of the communication is further handled by the *Passport* library [9] which can be used as an authentication middleware in the *Express* route definitions. If the LDAP server finds a match it returns an object with information about the user in question—if it is the first time the user logs in, the information is also saved in the database (the process is described in more detail in chapter 2). To confirm the authentication, the server creates a session and saves it in a *MongoDB*-backed session store. The session is then passed to the client where it takes the form of a cookie. From now on the client can use the cookie to automatically authenticate the user by sending it back. The server performs a lookup in the store to check if the given session exists and whether it has not yet expired. If both conditions are met, the user remains authenticated. The whole process is defined by a so-called strategy which can be seen in the figure 1.7.

```
passport.use('ldap', new LDAPStrategy(
  {
    server: {
      url: 'ldaps://ldap.fit.cvut.cz:636',
      searchBase: 'ou=People,o=fit.cvut.cz',
      searchFilter: '(uid={{username}})',
      searchAttributes: ['cn','uid']
    }
  },
  function(entry, done) {
    User.findByIdAndUpdate(entry.uid, { lastLogin: new Date() },
        function(err, user) {
      if (err) {
        return done(err);
      }
      if (!user) {
        var newUser = {
          _id: entry.uid,
          name: entry.cn,
          lastLogin: new Date()
        };
        User.create(newUser, function(err, user) {
          if (err) {
            return done(err);
          }
          return done(null, user);
        });
      } else {
        return done(null, user);
      }
    });
  }
));
```

Figure 1.7: *Passport* LDAP strategy

1.2.3.2 KOSapi

KOSapi allows us to traverse resources in the information system *KOS* but to access it, some kind of authorization is mandatory. To fulfill the requirement, yet another faculty service is used. This time, it is the OAuth 2.0 authorization server running at `auth.fit.cvut.cz/oauth`. The server provides different types of authorization, also known as grants. One of those are the so-called client credentials—these are obtained by registering the application in the authorization server application manager located at `auth.fit.cvut.cz/manager`. The application then sends an access token request to the endpoint located at `auth.fit.cvut.cz/oauth/oauth/token` along with the credentials. If granted, the token is saved and used to access *KOSapi* until the time it expires. When that happens, another token has to be requested.

1.2.4 Installation

In order to be able to install and run the application properly, several prerequisites are necessary. The obvious requirement is that the *Node.js* server application has been installed on the system. The package manager called *npm* comes automatically with the server and can be used to install further dependencies. This is done by calling `npm install` while in the root folder of the application—*npm* then looks in the `package.json` definition file and install all packages listed there into the `node_modules` folder. The packages are furthermore divided into two groups: those the application requires to run and those that are needed for development.

Another dependence is yet another manager called *Bower* [10] which is used solely for management of front-end packages. It can be installed using the command `npm install -g bower`. The `-g` flag ensures that the manager is installed globally and can be used from the command line. The required libraries are listed in the file `bower.json` and can be installed by typing `bower install` while in the root folder of the application. If successful, they can be found in the folder `public/lib`.

In order to continue with development of the application, another two requirements must be fulfilled. The first one entails installing *grunt* [11], a

JavaScript task runner that automatically performs repetitive tasks like minification of style sheets, concatenation of scripts, code linting, and so on. It can be installed by calling `npm install -g grunt-cli`. The other requirement is a locally running instance of *MongoDB* that can provide a database for testing purposes.

If properly installed, the application can be started by typing `grunt` in the root folder. A sequence of commands—defined in the `gruntfile.json` file— is then triggered and in the end, both the server and the application should be up and running. Whenever a change is made to any relevant files such as the style sheets or the server-side code, a set of tasks is performed—which tasks are run depends on the file in question—and if necessary, the server is restarted. If there is no need for further development and the goal is to simply start the server, running the command `node server/server.js` should suffice.

1.3 Client

...

1.3.1 Services

The core of the application lies in the so-called services which are objects that are used to organize code into logical units and which are wired together using dependency injections to create new functionality. They are built upon services already present in *AngularJS* like `$http` or `$resource` and are instantiated only when another component depends on them. The so-called service factory then generates a single instance and provides a reference. Any other component that might require the same service will receive a reference to the same instance. Using this feature, data can be easily shared between different parts of the application.

Much of the core functionality is handled by aptly named services such as `ExamTake` or `NewBlueprint`. The user interface controller actually depends on many of these, in order to provide information about their state to users. It is for these reasons that most of the services get instantiated right at the initial load. Nonetheless, some of them are used only by a subset of users,

depending on the user role. That is why we want to know the role before creating instances of services a user might not be able to use anyway—and because the majority of the services also create event listeners, which would only pose as potential memory hogs.

This is where the User service comes in play. It is able to tell if a user is logged in and what is eventually his role and then provide that information to other components. When a user successfully authenticates, the service emits a loggedIn message to $rootScope—kind of a common ground for all components that depend on it. Any service listening will receive the message and inspect the value of the role in return. Knowing that information, it can safely decide whether to self-initialize or not.

1.3.1.1 Persistence

As mentioned before, the services are used to hold data among different states and controllers. But if not taken care of, the moment the browser refreshed or reloaded the page, all that data would be lost and services would have to initialize afresh. To handle this situation, a beforeunload event listener is attached to the window object. Whenever the page is closed or reloaded, the listener emits a save message to $rootScope. Any service already holding data responds by serializing the data using the JSON.stringify() function and saving the result into localStorage—if not supported, several fallbacks are tried before saving as a cookie. While initializing, each service then peeks in the storage first and if a previously saved state is found, it is loaded and deserialized. If not, the service continues normally.

1.3.2 Directives

Directives are what could be considered *AngularJS's* answer to Web components. They are composed of HTML templates and logic, encapsulated together to form small and reusable units, camouflaged to the DOM either as custom elements or attributes. They can be set up in a way that makes their innards visible to the outside world. Most of the time, though, they create what is called the Shadow DOM, having their own isolate scope (and $scope).

Directives also provide a rare occasion for a direct interaction with the DOM—otherwise a forbidden fruit in *AngularJS*.

In the application, directives are used mostly in the form of custom elements that uses native HTML elements for their structure and add logic to support manipulation with their contents. In most cases the interactivity is easily achieved by adding the attribute `contenteditable`. A problem lies in the fact, that unlike `input` elements which natively support *AngularJS* two-way binding through the `ng-bind` directive, other elements do not bind to a model, even if made editable in this way. Fortunately, this can be solved by using another custom directive that is applied alongside the `contenteditable` attribute.

The directive can be seen in the figure 1.8. It uses the internal `ngModel` service to replicate the native binding mechanism. It also attaches an event listener that is triggered the moment a user tries to paste data into the element. Normally, any style associated with that data would be passed along as well—a highly undesirable behavior at best. The event is therefore hijacked and implemented in such a way that the data is taken directly from the clipboard as a plain text and inserted manually.

Such solution allows us to bind any content to one of the custom elements through a public interface and watch it update automatically whenever a change occurs. Nonetheless, given the distinct nature of the elements, each requires a different type of content. The most simple of these are the two directives `paragraph` and `snippet`—the first one being self-explanatory and the second serving as a container for fragments of code. They are similar in that they both expect a string for the content. Apart from that, the `paragraph` does little more than the native `p` element. Its purpose is mainly to unify the non-uniform behavior of the `enter` key among browsers—some create a newline by means of the tag `br`, others by adding `div`.

1.3.2.1 Snippet

The `snippet` directive, on the other hand, does much more. Besides content, it is given a variable containing a name of a programming language, such as JavaScript or C++. Using an external library called *Prism*, the content is then

```
app.directive('contenteditable', function() {
  return {
    restrict: 'A',
    require: '?ngModel',
    link: function($scope, $element, attrs, ngModel) {

      $element.on('input', function() {
        $scope.$apply(function() {
          ngModel.$setViewValue($element.html());
        });
      });

      $element.on('paste', function(e) {
        e.preventDefault();
        document.execCommand('InsertHTML', false, e.clipboardData.
            getData('text/plain'));
      });

      ngModel.$render = function() {
        $element.html(ngModel.$viewValue || '');
      };

    }
  };
});
```

Figure 1.8: Directive for editable elements

parsed according to rules specific to the language and any relevant keyword is wrapped with a span tag and a class that further specifies the node. Highlighting the content is therefore only a matter of applying the right styles. Every time the content changes, this procedure is repeated. Furthermore, the element is given an event listener that watches for a tab key being pressed. When that happens, the default behavior—changing the focus to the next element—is prevented and the content is instead indented. As expected when working with code, the depth of the indentation is maintained even if the user enters a newline.

1.3.2.2 List and options

The directives list and options both require the content to be an array of objects. In the case of list, an array of strings would normally suffice, at least

19

```
$element.on('keydown', function(e) {
  var k = e.keyCode;
  if (k === 13 || k === 8 || k === 46 || k === 38 || k === 40) {
    var item = e.target;
    var items = getElements('li', $element[0]);
    var index = items.indexOf(item);
    if (k === 13) {
      e.preventDefault();
      $scope.items.splice(index + 1, 0, { content: '' });
      $scope.$apply();
      getElements('li', $element[0])[index + 1].focus();
    } else {
      var focusOn = null;
      if ((k === 8 || k === 46) && $scope.items.length > 1 && !
          $scope.items[index].content) {
        $scope.items.splice(index, 1);
        $scope.$apply();
        focusOn = items[index - 1];
      } else if (k === 38) {
        focusOn = index > 0 ? items[index - 1] : null;
      } else if (k === 40) {
        focusOn = index < items.length - 1 ? items[index + 1] : null
            ;
      }
      if (focusOn) {
        e.preventDefault();
        var range = document.createRange();
        range.selectNodeContents(focusOn);
        range.collapse(false);
        var selection = window.getSelection();
        selection.removeAllRanges();
        selection.addRange(range);
      }
    }
  }
});
```

Figure 1.9: Use of event listeners in directives

if one only needed to show the contents. But we must be able to modify it too, and the directive ng-repeat—used to traverse and display the items—requires a list of objects to be able to tell which item is which whenever a change occurs.

It is also necessary to provide a way to create and delete items from the list. This is achieved by binding a key press event listener to the element. The

listener makes the key `enter` and `backspace` create or delete an item, respectively. In the first case, the new item is created after the current one and the focus is changed accordingly. In the second case, the item is removed only if the cursor is at the beginning of its content or if empty. The listener in question can be seen in the figure 1.9. While the `options` directive is a little different because it also holds a boolean value besides the content, it essentially behaves the same way.

1.3.2.3 Canvas

The most complex among the directives is the `icanvas`—a container around a `canvas` element that provides tools for editing the content. The directive is in fact not interacting with the canvas itself, but instead with a high-level abstraction that encapsulates the native low-level API the canvas provides. This is made possible by the external library *Fabric.js* [12]. Apart from giving us a more efficient way to manipulate the canvas' contents, the library also allows us to serialize the canvas either into a JSON object or a SVG image— and deserialize it back into a canvas if needed. This is essential if we want to store the resulting image into the database.

The tools provide the means for drawing basic shapes on the canvas and manipulate them as needed. This is achieved by registering a set of listeners that operate on mouse events. Whenever a tool is activated, all event listeners previously applied to the canvas are removed and a new set, distinct for each of the tools, is added instead. An example of a tool handler can be seen in the figure 1.10.

If no content is given, the directive creates an object containing an empty array and an index set to zero. A clear canvas is then initialized and serialized into a JSON object using the library. This object is pushed into the hitherto empty array and after that, the whole package is fed to a `History` object, which represents the various states of the canvas. The moment the canvas is modified, it is once again serialized and the results get pushed into the array of states as before, while the index is raised by one. This essentially allows us to perform undo and redo operations on the canvas—both of which are provided through the public interface of the `History` object. The user can

```
var circleHandler = {
  mouseDown: function(o) {
    canvas.calcOffset();
    isMouseDown = true;
    pointer = canvas.getPointer(o.e);
    obj = new fabric.Ellipse({
      strokeWidth: 1,
      fill: 'rgba(255, 255, 255, .25)',
      stroke: 'rgba(0, 0, 0, .75)',
      top: pointer.y,
      left: pointer.x,
      originY: 'top',
      originX: 'left',
      selectable: false,
      rx: 1, ry: 1,
      perPixelTargetFind: true
    });
    canvas.add(obj);
  },
  mouseMove: function(o) {
    if (!isMouseDown) {
      return;
    }
    var currentPointer = canvas.getPointer(o.e);
    obj.set({
      rx: Math.abs(currentPointer.x - pointer.x),
      ry: Math.abs(currentPointer.y - pointer.y)
    });
    canvas.renderAll();
  },
  mouseUp: function() {
    isMouseDown = false;
    obj.set({
      width: obj.rx * 2,
      height: obj.ry * 2,
      top: obj.top - obj.ry + 1,
      left: obj.left - obj.rx + 1
    });
    canvas.renderAll();
    canvas.fire('object:modified', { target: obj });
  }
};
```

Figure 1.10: Example of `icanvas` tool handler

```
api.undo = function() {
  if (api.canUndo()) {
    canvas.getObjects().forEach(function(obj) {
      canvas.remove(obj);
    });
    canvas.loadFromJSON($scope.content.states[--$scope.content.
        currentState]);
    if (!canvas.selection) {
      canvas.getObjects().forEach(function(obj) {
        obj.set({
          selectable: false
        });
        obj.setCoords();
      });
    }
    canvas.renderAll();
    $scope.$apply();
  }
};

canvas.on('object:modified', function() {
  if (content.currentState < content.states.length - 1) {
    content.states = content.states.slice(0, content.currentState +
        1);
  }
  if (content.currentState < limit - 1) {
    content.currentState++;
  }
  if (content.states.length === limit) {
    content.states.shift();
  }
  content.states.push(angular.toJson(canvas));
  $scope.$apply();
});
```

Figure 1.11: Example from History module

traverse the states back and forth, unless he makes a change to the canvas while in the middle of the list. When that happens, the array's tail is cut and with it all the states that were further down the list.

If the directive instead receives a content in the form of a SVG image string, it is simply parsed by the library and converted into a canvas. The content object is then transformed into the structure described above, only this time, the first state is not clean but contains the provided image.

23

1.3.3 Routing

...

1.3.4 Style sheets

The look and feel of the application is described using CSS style sheets. While fine by itself for small applications, CSS is highly unsuitable for more complex projects, due to its inherent inability to create modules. Even though such thing is actually possible through the CSS function `@import`, the performance drawbacks are so severe that it stops being a viable option. To supply this need, some kind of CSS preprocessor therefore must be used.

1.3.4.1 Preprocessor

Sass [13] is the most natural choice, being the de facto solution for construction of complex style sheets. In contrast to vanilla CSS, the *Sass* `@import` function is not called on runtime but during the compilation process. This allows us to create highly manageable structure of self-contained modules, using the principle of separation of concerns. Each of these modules can deal with different aspect of the presentation—one may be used to normalize various peculiarities among browsers, another can focus solely on typography or forms, and so on. Using *Sass* also gives us the opportunity to employ variables, rule nesting, mixins, or selector inheritance—none of which are available in vanilla CSS.

1.3.4.2 Methodology

When it comes to structuring the style sheets, it is always a good idea to choose some kind of methodology and stick to it if possible. *BEM* [14]—meaning block, element, modifier—is a front-end methodology that introduces a way for naming CSS classes to give them more transparency and meaning. It follows the pattern shown in the figure 1.12, where `.component` represents some sort of higher level abstraction, `component__element` refers to its descendant that helps form it as a whole, and `.component--modifier` stands for a different state or version. The first block in the figure is written in *Sass* notation while the second one shows what would be the output in vanilla CSS.

```
.component {
  &__element {}
  &--modifier {}
}

.component {}
.component__element {}
.component--modifier {}
```

Figure 1.12: *BEM* methodology pattern

The strength of *BEM* lies in the fact that anyone reading the markup can easily tell the relationship between different elements and does not have to guess by judging the structure. Yet another reason is performance. The selector engine parses a query from right to left which means that if given for example the selector `.site-nav a`, it first matches all a elements on a page. For each of those, the engine then has to traverse up the node tree to find out whether a parent with the class `site-nav` exists—using the selector `.site-nav > a` can help but is not always possible. If, on the other hand, the a elements are given the class `site-nav__item`, the selection process is reduced to a simple lookup of elements matching the class.

The drawback of this methodology is the considerable increase of class declarations in the document structure, which in turn leads to larger files. However, the overhead is greatly reduced when serving files using a GZip compression—the often repeated class names tend to compress well.

1.3.4.3 Breakpoints

A grid system is almost always a necessity when constructing a web layout. While CSS offers its own grid layout module, the support among browsers is unfortunately almost nonexistent. It is therefore necessary to build one's own solution. Notwithstanding the final approach, any grid system should be responsive to changes in viewport width. This can be done by creating classes that set a grid's item to different widths depending on the currently active media query—a conditional block that is defined by a set of constraints,

also known as breakpoints. The breakpoints refer to the viewport width and are device-agnostic, which means they are not constructed around specific devices and their resolutions, but instead, around the natural breakpoints of the layout itself.

```
$breakpoints: (
  'portable' '(max-width: #{($layout-width - .001)})',
  'considerable' '(min-width: #{$line-length})',
  'palm' '(max-width: #{($line-length - .001)})',
  'lap' '(min-width: #{$line-length}) and (max-width: #{($layout-
     width - .001)})',
  'desk' '(min-width: #{$layout-width})'
);

body {
   animation-name: none;
   @each $breakpoint in $breakpoints {
      $type: nth($breakpoint, 1);
      @include media-query($type) {
         animation-name: #{$type};
      }
   }
}
```

Figure 1.13: Breakpoints definition in *Sass*

Even though the breakpoints, defined as shown in the figure 1.13, work well while used in the style sheets, sometimes we need to employ media queries in JavaScript, too. For example, the client might decide to load certain piece of code only if the screen is large enough, in order not to waste a user's bandwidth. While JavaScript supports media queries on its own, to use it means facing a different kind of problem: we would have to define the breakpoints on two places and therefore repeat ourselves. To solve this conundrum, a trick is used. The body element is given an `animation-name`—harmless on its own, unless an animation with the same name is found. The property is then filled with the different breakpoint string identifiers, depending on the currently active media query. All that is left to do is to define a function in JavaScript that returns the current value of the `animation-name` property—this is achieved by calling `getPropertyValue('animation-name')` on the computed-

style object of the body element. Using this technique, the code remains DRY and the active media query is always within reach. An even better solution would be to utilize the content property of a pseudo-element attached to the body—unfortunately, *Google Chrome* currently cannot retrieve this value using JavaScript only.

1.3.5 Optimization

Front-end optimization refers to a set of techniques that are used to enhance the performance and efficiency of any application running in browser. These include minifying the style sheet, reducing the amount of HTTP requests, the concatenation of scripts, the so-called lazy loading of resources, and many others.

The main bottleneck of browsing speed, as opposed to video streaming, is not network bandwidth but latency [15]. Because of this and due to the inherent qualities of the HTTP/1.1 protocol, it is crucial to reduce the number of HTTP requests on the page by any means necessary. This will hold true until the new version of HTTP—based on the Google's *SPDY* protocol—receives widespread adoption.

The first step usually entails concatenating all JavaScript libraries and client-side code into one file and minifying the result. Because loading an external JavaScript is a blocking operation—unless loaded asynchronously—it is a good idea to place the script element that refers the resulting code bundle as the last item of the body element. That way, the HTML and CSS can start loading first.

In order not to waste bandwidth, any code that is not essential for running the application is loaded only on demand. This is for example the case for the *Fabric.js* library—downloading 204 kB of possibly unnecessary code might be fine while browsing on a home network, but not so much on a mobile data plan.

And although minification helps considerably reduce the size of a style sheet, a more drastic measure can be taken in order to scale it down even further. While indispensable, grid systems tend to require a great amount of class definitions, most of it never used. There is therefore no harm in remov-

ing the ones not used from the style sheet—the same applies for any other needless CSS declaration. The process can be automated by loading the application in *PhantomJS*, a WebKit-based headless browser that supports DOM handling and CSS selection. Each declaration in the style sheet is tried against the DOM and if no match is found, the declaration is omitted. Using this technique, the size of the style sheet can be reduced by more than 50 %.

1.4 Interface

1.4.1 Design

Taking the proposed use of the application into account, it is crucial to embrace the idea that the design might be the most important aspect of the product and also the ultimate measure of its reach and usefulness. And when talking about design, we mean not just the aesthetics, of course, but also the whole field of user experience and usability. It is no secret that dealing with a poorly designed product can actually become a source of stress by itself, or just a very unpleasant experience at best. The situation one finds himself in while trying to use such application usually only helps to heighten this feeling—and taking an exam can be quite gruelling as it is, as anyone can attest. Forcing students to use an application that is hard or simply obnoxious to use would epitomize nothing more than a major tell-tale sign of bad decision-making at every level of the design process.

1.4.1.1 Flat or skeuomorphic?

As Dieter Rams once famously stated in his ten principles for good design (sometimes understandably referred to as the "Ten commandments"), good design is as little design possible [16]. And even though Rams dealt mostly with product design, the statement is applicable everywhere. Less, but better—such design can concentrate on the essential aspects instead of burdening the product with non-essentials. In recent years, this minimalist approach has once again proved to be in good health when the so-called "flat" design seized the reins of user interface design everywhere. The reason it is referred to as

flat is because it removes any stylistic choices that give the illusion of three-dimensionality (such as drop shadows, gradients, or textures) and focuses on simple elements, typography and primary colors.

In opposition to flat design lies what is called skeuomorphism. Essentially any interface using ornamental cues referring to an object in the physical world is said to be skeuomorphic. These cues are intended to help the user navigate the interface by creating a sense of familiarity. Among the many arguments against skeuomorphic design are those which state that skeuomorphic interface elements use metaphors that are more difficult to operate and take up more screen space than standard interface elements or that many users may have no experience with the original device they emulate. Also prevalent is the opinion that such design limits creativity by grounding the experience to physical counterparts.

Valid or not, these arguments helped pave the way for the domination of flat design. One does not have to look far for evidence—take for example the design of *Windows 8* and compare it to previous versions. Then do the same with *iOS 7*. The trend's existence is indisputable—but it is the author's belief that as with any trend, it is best not to get carried away and trust one's judgement first. It is undeniable that flat design allowed interface designs to be more streamlined and efficient, and to quickly convey information while still looking visually appealing and approachable. However, when used without questions and reservation, it can actually obfuscate the meaning behind an interface.

This is what Dieter Rams meant when he said that good design makes a product understandable. We can take the cornerstone element of every interface—a button—and use it as a kind of guinea pig. If we blindly followed the principles of flat design (which many do), the element's visual identity would quickly fall apart. But say we add a simple drop shadow—the metaphor then holds and so do the streamlined aesthetics. The point is that the button does not have to look exactly like a button to be recognized as one. Giving the user a cue is usually enough, his mind will do the rest and fill the gaps. The same applies for an input element—the user expects it to have at least a slight inset drop shadow and is confused when the element is not given
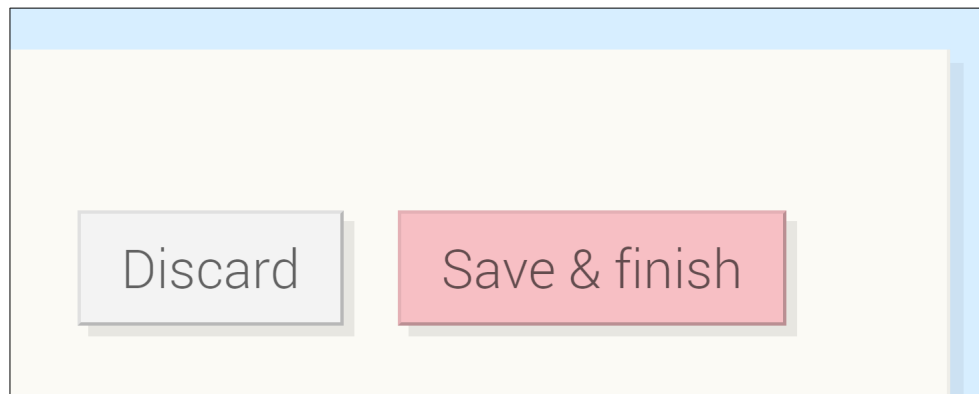
29

Figure 1.14: Example of buttons

one. This has been proven time and again by many usability studies.

It is because of these and many other reasons that the approach taken in designing the application user interface, while based on the principles of flat design in its focus on typography and color, is not shying away from some of the more reasonable aspects of skeuomorphism. Even though the design does rely heavily upon simple shapes and deliberate use of color, the most ostentatious attempt at emulating reality is immediately apparent when one takes a look at the page layout, as seen in the figure 1.15. The main content is presented in a form reminiscent of a sheet of paper. This is heightened by the use of color, a subtle drop shadow and the fact that when there is not enough content to stretch it further, the "sheet" holds a ratio of 1:1.414, the same as a real piece of paper. The reasoning behind this idea is to provide the users—especially the students—with something familiar. Novelties and surprises have their place in design, but one must choose wisely how and when to introduce them—confounding a student who is probably already struggling with his exam is hardly a good idea.

The side effect of this decision takes shape in the relatively small portion of the available viewport width being occupied by the main content, at least on displays designated for desktop use. This further strengthens the idea that the main content should always be in the spotlight. Any additional information that might not be essential but proves helpful is placed outside of this
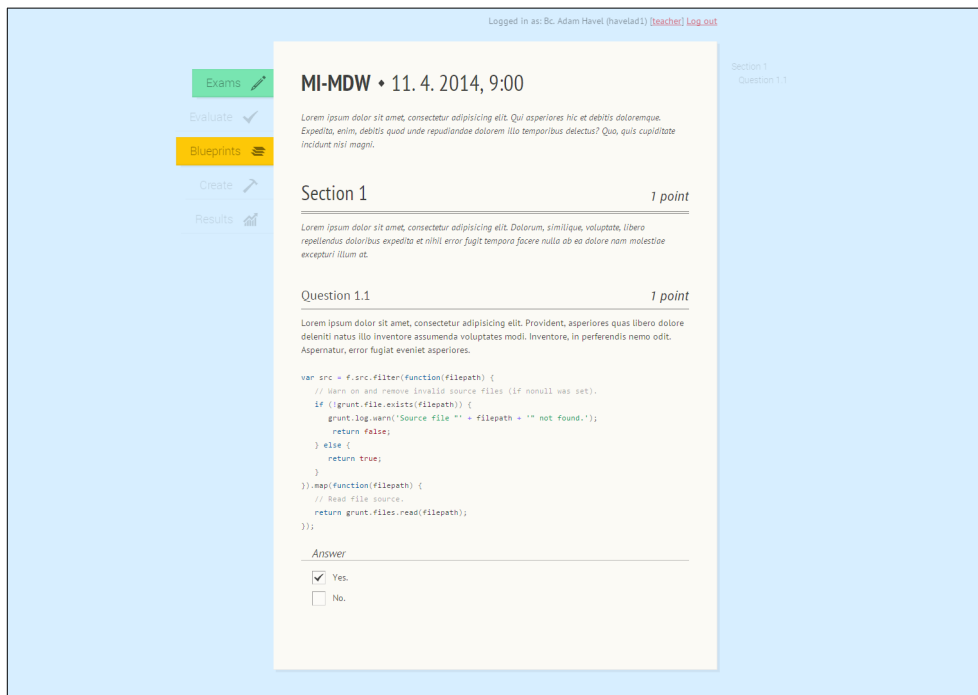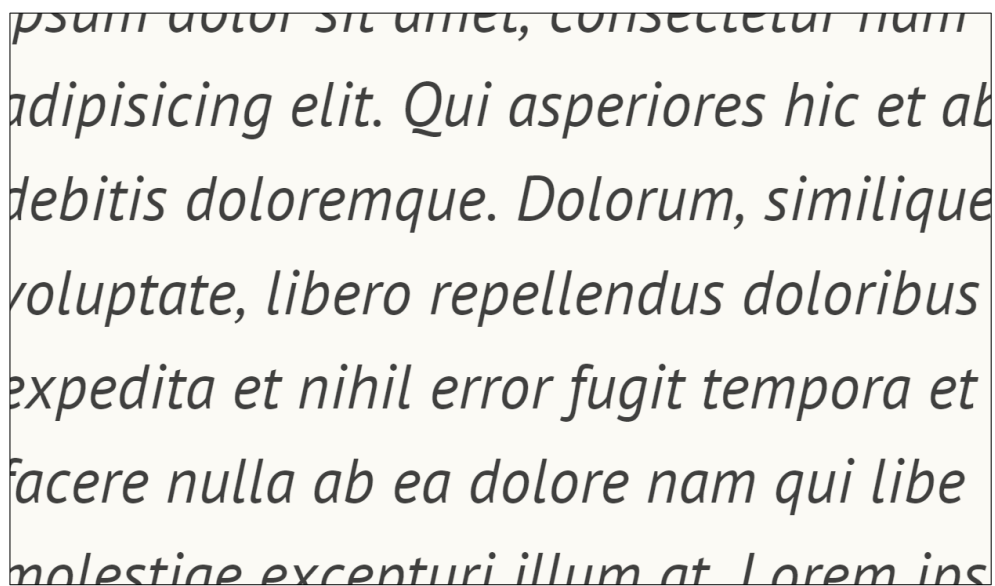
Figure 1.15: Page layout

area—its opacity reduced unless in focus, in order not to distract the user. An example of such a component is the table of contents which apart from providing the document structure allows an user to tell his current position and to swiftly jump to a different part by way of anchor links.

The main navigation menu behaves similarly—translucent until the moment it is actually used. Only the active items are highlighted continually by taking on a likeness of bookmarks. Not just an aesthetic choice, this application of skeuomorphism is also meant as way to get across the idea that just like with a real book, there can be more than one bookmark at a time, each representing a shortcut to some point of interest.

1.4.1.2 Typography

Notwithstanding the importance of aesthetic qualities, in the end, what matters the most is the actual content. In this case, the content is mostly textual, it is therefore necessary to make educated choices about the way the

> *psum dolor sit amet, consectetur nam*
>
> *adipisicing elit. Qui asperiores hic et ab*
>
> *debitis doloremque. Dolorum, similique*
>
> *voluptate, libero repellendus doloribus*
>
> *expedita et nihil error fugit tempora et*
>
> *facere nulla ab ea dolore nam qui libe*
>
> *molestiae excepturi illum at. Lorem ips*

Figure 1.16: Example of *PT Sans* typeface

text is presented and structured. Such is the focus of the art of typography—arranging type in order to make the language it forms most appealing to reading and recognition. This involves the selection of typefaces, choosing the right font sizes and weights for each situation, adjusting the length and spacing of line, and much more.

The first and probably most important decision is to pick the right typeface—one that hopefully reflects the qualities the product is trying to represent. In the case of this application, the choice came upon *PT Sans*, a freely distributed font family that was released under the *SIL Open Font License*. It was developed by a foundry called *ParaType* under the sponsorship of the Russian government, as a way to provide a modern font that encompasses all characters contained within the many languages of Russian federation [17]. It is a professional-grade typeface that combines traditional and conservative appearance with modern trends of humanistic sans-serif, and one that possesses an enhanced legibility. While choosing a serif font family would make more sense for longer passages of text, in this case the content is expected to consist of rather short snippets, sans-serif family therefore makes a better choice and matches nicely with the streamlined aesthetics of the application.
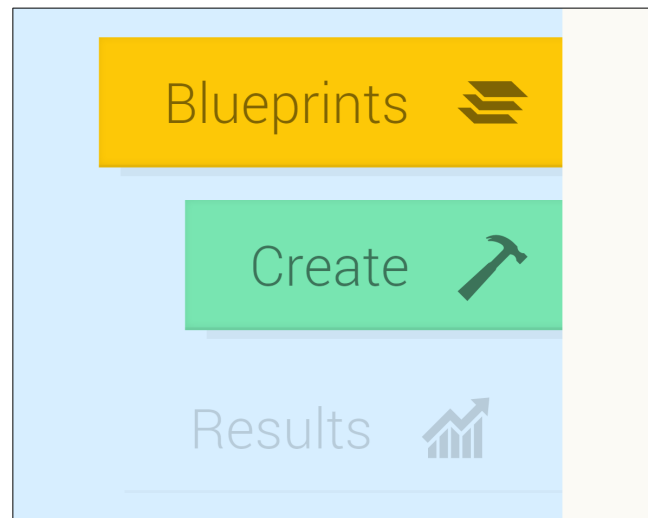
Figure 1.17: *Roboto* typeface as used in the interface

When combining different typefaces, it is necessary to pick font families that share similar x-height, which is—not very surprisingly—the height of the letter *x* in a given font. If one does not feel adventurous, it is generally a good idea to choose typefaces from the same superfamily. The result is a set of fonts that, while belonging to different classes like sans and serif, have a similar appearance and properties, such as the aforementioned x-height. It is for these reasons that *PT Sans Narrow* nicely complements the body text typeface while used in headings—the contrast such combination creates helps further define the document structure visually.

The same rationale has been used when selecting the *PT Mono* typeface. It is a monospaced family which means the font's letters and characters each occupy the same amount of horizontal space. That makes it appropriate to use when displaying fragments of computer code or for numerical tabular data.

Last on the list is *Roboto*—a typeface developed by Google to serve as the default font family for the Android operating system. While often described as a "Frankenstein among fonts", because of the way it mixes different typographic styles, it is actually a good match for interface elements. It is therefore used in the navigation menu, and all the buttons and prompts.

Choosing the right font sizes for different contexts is usually next on the

list and should not be done haphazardly—the whole layout of the page is (or should be) affected by the typographic decisions made, not the other way around. This is true for two reasons: first, the layout width should be set in such a way that any given line of text does not contain much more than around 70 characters—the amount which is believed to provide the best legibility. And second, all layout dimensions should be expressed in a relation to the base font size—this actually makes the first point trivial to achieve and is the raison d'être of the em unit. The base font size is however best left set at the default value of 1 em, which usually renders as a font size of 16 pixels. The user can then decide herself whether to enlarge or reduce the font size and changing thus proportionally the rest of the layout.

The moment we want to add different font sizes to our mix, we find ourselves at a crossroad. We can travel the path of arbitrary, conventional or easily divisible numbers, or we can use the so-called modular scale [18]. Making such scale is quite straightforward—a set of values is generated by choosing a number and a ratio, and multiplying or dividing the first with the other. In our case, the number is the base font size. As for the ratio, any proportion might do, but it is probably a good idea to pick one that is rooted in geometry, music, nature or history. Such ratios are culturally relevant and meaningful, and also tried and proven by centuries of use. More importantly, they tend to produce visually pleasing compositions.

The golden mean with its proportion of 1:1.618 is an obvious choice, but there are many others, equally useful. One of the simplest is the musical fifth with a ratio of 3:2. When put into our scale, it generates a value of 1.5 em— a good starting point for line spacing, which will be covered in a moment. Nonetheless, we might find that our set of values is too sparse and does not cover all needs. The remedy is often to create a double-stranded scale, one that is based not on one but two numbers. Like the first one, the second number should not be arbitrary—line length, which in our case equals 42 em, is usually a good choice.

Another essential property is the already mentioned line spacing, also-called leading. Using a value from the modular scale, it is set to be one and a half times of the font size. To further improve legibility, the leading of the

```
@mixin blank-lines($lines: 1, $direction: trailing, $type: margin) {
  $space: $lines * $leading;
  @if $direction == trailing {
    #{$type}-bottom: $space;
  } @else if $direction == leading {
    #{$type}-top: $space;
  } @else if $direction == both {
    #{$type}-top: $space;
    #{$type}-bottom: $space;
  }
}
```

Figure 1.18: Vertical rhythm in Sass mixin

body text is then chosen as the base for what is called the vertical rhythm—a principle of unifying the line heights of different elements so that each line, blank or not, can be expressed only as a multiple of this base. It is important to note that the base is no longer relative, i.e. expressed in em, but absolute. In the case of our body text with a font size of 16 pixels and line height set to 1.5 em, the base holds a value of 24 pixels. Multiples of this number are then used to define any line height, spacing or dimensions in the vertical direction. This is reflected in the style sheets, where except a few cases, every vertical margin or padding is expressed using the Sass mixin shown in the figure 1.18. The multipliers do not necessarily have to be integers but they should sum to a whole number. It is therefore allowed to have for example an element with margin of half the leading in both directions, since it adds up to one.

1.4.1.3 Colors

The backdrop color has been chosen deliberately for its properties and effects on human mind. A light that is composed predominantly of the color blue has a direct and powerful influence on the secretion of melatonin, a hormone that regulates sleep patterns in mammals [19]. So while it is generally a good idea to avoid blue light at night in order not to disturb one's sleep, the very same quality can be used during the day to effectively raise a person's alertness [20].

While other colors might not have the same powerful properties, each of

```
$colors: (
  blue: #d7eeff,
  paper: #fbfaf5,
  yellow: #fdc807,
  red: #d1192b,
  green: #78e5b1,
  purple: #957bf8
);
```

Figure 1.19: Definition of colors in *Sass*

them holds some kind of meaning, which is more often than not defined a culture. It would be therefore unwise not to follow these leads and use the colors haphazardly. One simply has to know the product's target market and adjust accordingly to local traditions. Little details such as the fact that in the Japanese culture, the color white represents death may seem peculiar for us Westerners, but in the end do matter.

As already mentioned, the main content is filled with an off-white color expressed as a RGB hex triplet with a value of #fbfaf5. Apart from trying to mimick the properties of a real paper, such color tends to be easier on eyes than pure white. The rest of the colors represent what is often called the brand of a product. They should be distinct enough so as to be immediately recognizable. In many cases, they are also employed as a kind of a visual cue to convey different meanings. These should be easier to grasp than for example a written prompt by itself—at least for those users that do not suffer from colorblindness. This approach is used for example in the navigation menu, as seen in the figure 1.20: a differently colored bookmark represents a different state of a given item. The yellow bookmark (#fdc807) stands for the currently active page. The green one (#78e5b1) hints at a work in progress. And when a bookmark turns red (#d1192b), it is to draw attention to an urgent matter, for example exams that have yet to be evaluated.

To maintain a coherent visual identity, only the colors stated above and their variations are used throughout the application. The variations are created by means of two internal Sass functions: ligthen() and darken(). These

Figure 1.20: Use of colors in the menu

functions take an original color as the first argument and an amount of change expressed in percentage as the second. The result is a color that is not different in hue from the original one—only the amount of white is augmented or reduced.

Anyhow, everything written about colors so far holds especially true for the so-called modal windows. These are utilized heavily in the application and can be used as another example to further expand upon the argument for the importance of colors.

1.4.1.4 Modals

A modal is a type of a child window that requires a user interaction in order to allow the user to return to the main window. They are used to draw attention to vital pieces of information or to display emergency states. And even though they are often criticized for breaking the user's workflow, they are irreplaceable in many situations. Most of their shortcomings can be remedied by using simple techniques such as darkening the background behind the window in order to focus the user's attention.

Another important aspect of modal windows is the fact that prompts tend to be either read wrong or not at all. It is therefore necessary to provide some countermeasures, in order to minimize the chance a user will do something
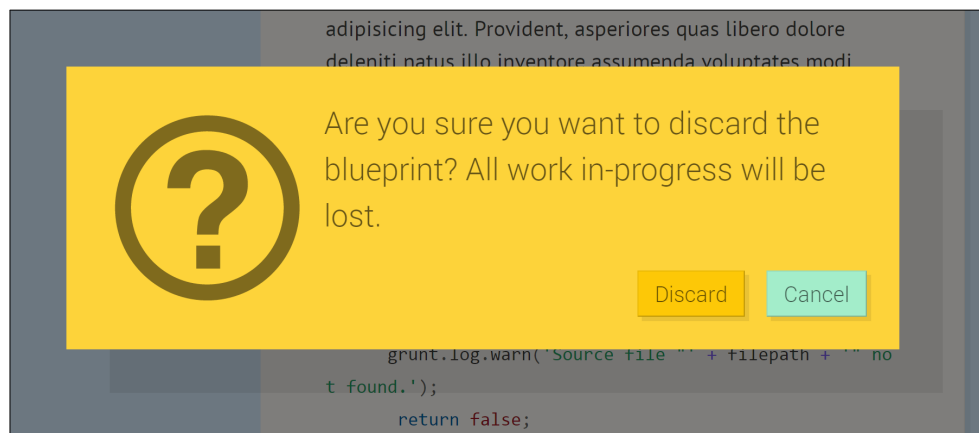
Figure 1.21: Modal window

he might regret later. There are several thing that can be done: first, as already stated before, it is important to create a consistent visual identity and adhere to it strictly. By using the color cues, a user is able to tell the general meaning behind a prompt, even before reading the message. And there is no need to go against the grain—an error message should be red and confirmation of success green. Yellow is meant to alert the user about a non-recoverable action he is about to make or in general any event that requires an immediate attention. Last but not least, purple holds the message that some kind of input—like submitting an image—is required in order to continue.

Furthermore, if possible, the user must be given a way to abort the action in question. This is usually accomplished by means of a cancel button. If present, it is also necessary to tell what is the primary action for given prompt. For example, when a modal warns the user he is about to delete something, the primary action is to cancel the removal. This is expressed in several ways. Visually, by emphasizing the cancel button and making the other, so-called secondary action button less pronounced. Positionally, by placing the primary action button as the rightmost, which is the place the user is most likely going to click. And last, functionally, by binding it to the enter key.

Processes

2.1 Authentication

No part of the system is accessible without authentication. Upon loading, the client makes a `GET` request to a resource defined at `/api/user` and appends a session identifier if a cookie is found. The server then tries to look up the identifier in MongoDB-backed session store. If successful, it checks whether the session has not expired—if that is not the case, it sends the client a JSON object containing information about the particular user. The information is obtained by deserializing the user identifier from the session and using it to fetch relevant data from the database. The client application then fills its `User` service with the received object and the user is onward recognized as authenticated.

2.1.1 Logging in

In the opposite case—either no session was found or it has already expired— the server responds with a 401 status code which forces the client to show the user to a login form. After filling in the form, the client makes a new request to the `api/user` resource, this time using the `POST` method, sending along user credentials. These credentials are then checked against the faculty LDAP server running at `ldap.fit.cvut.cz` using a secure connection. If user's identity is verified, the LDAP server responds with basic information about the user. Back at the application server, the database is queried for ad-
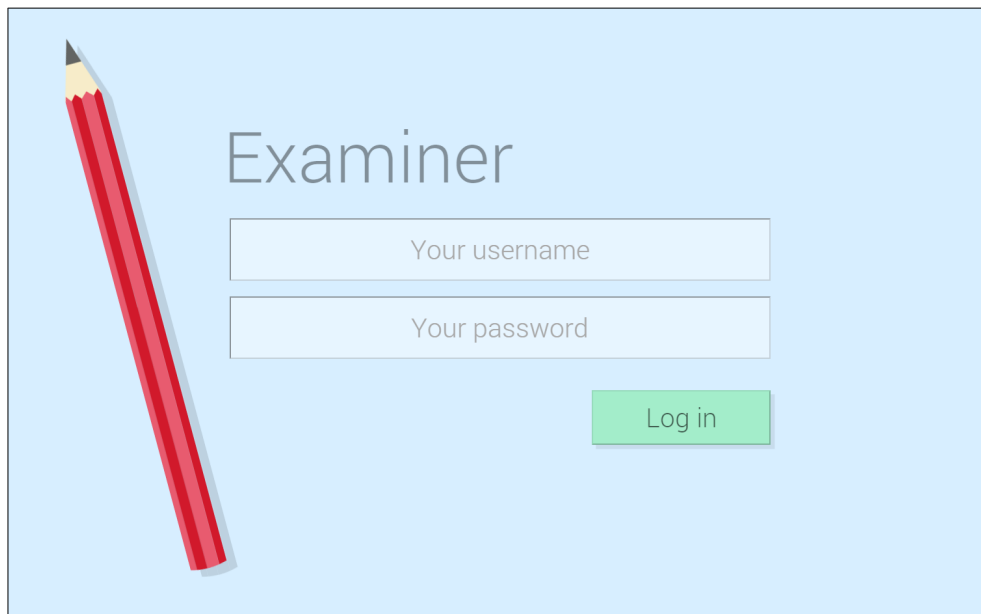
Figure 2.1: Login form

ditional data using the user identifier. It either finds a relevant entry or not. In the former case, the entry is updated with a new timestamp, representing the last login time, and the server sends it to the client as a JSON object, the same as before. If the latter is the case, it means that the user is logging in for the first time.

A new instance of `User` model is then created and filled with the available data. What remains unknown is the user's role—that is, if he or she is a student or a teacher. That information can be obtained by making a request to *KOSapi*, a faculty service that provides a REST API over the university information system *KOS*. To find out the role of a person with a given user identifier we can utilize the resource `/people/{uid}`. With that issue resolved, we can continue by looking up all courses the user either studies or teaches by using the resources `/students/{uid}/enrolledCourses` and `/teachers/{uid}/courses`, respectively. It is worth noting that *KOSapi* returns the data formatted as XML, which means the server has to transform it into a JSON object, for which it uses an external library called *x2js*. After obtaining all the necessary data, the new user can finally be saved in the data-

base and sent to the client.

But since both the role and the subjects of a user can change, the server has to periodically request the current states of the aforementioned resources and make necessary changes in the application database. This is one of the reasons the last login time is saved in the `User` model—using this information, the server makes sure not to ask for new data more than once a day. And since the majority of these changes occur at the start of a semester, long before any exam takes place, it should be frequent enough for most cases.

2.1.2 Client-side verification

When the client receives the authentication data and stores it in the appropriate service, the user can start to use the application with the scope assigned to his role. If for some reason the `User` service gets unset again, the user is shown a login form and any navigation elsewhere is disabled until he successfully logs in. In this manner, the authentication is verified only via the client-side service which might not be up to date with the server. This is sufficient for some operations and serves to mitigate any latency that would arise from communication between the client and the server.

If the user logs out, the `User` service is emptied and the client sends a `DELETE` request to the `/api/user` resource which prompts the server to discard the session. Nonetheless, the client session might also be terminated at the server end, either by force or because it has expired. It is therefore necessary that the client recognizes this situation and acts accordingly. This is ensured by using a HTTP interception service that monitors every response the client receives from the server when it tries to reach a resource. Whenever a 401 status code shows up, it means the session has been ended and the user has to authenticate again if he wishes to continue.

The HTTP interceptor can be seen in the figure 2.2. It requires the `User` service so that it can tell whether a user is present and to log him out if necessary. In any other type of component, the `User` service would be injected normally, but that is unfortunately not possible in this scenario. The problem arises from the fact that injecting a service into another component makes it dependent on that service. And since the `User` service is already dependent

```
app.factory('AuthInterceptor', function($q, $injector, Modal) {

  return {
    'response': function(response) {
      if (response.status === 401 && $injector.get('User').data) {
        Modal.open('alert', 'Your session has been ended. You will
            have to authenticate.', null, 'Log in');
        $injector.get('User').logout();
        return $q.reject(response);
      } else {
        return response || $q.when(response);
      }
    }
  };

}).config(function($httpProvider) {
  $httpProvider.interceptors.push('AuthInterceptor');
});
```

Figure 2.2: HTTP Interceptor

on $http and applying the interceptor essentially modifies the instance of the very same $http, making $http dependent on the User service through the interceptor would create what is called a circular dependence. That can be remedied by using the built-in $injector service which is able to get hold of another service reference without creating a dependence.

Something similar to the 401-filled responses also occurs when a user is not authorized to view a selected resource. This applies for example in the case when a user with the role of a student requests a different student's results. When such a thing happens, the server responds with a status code 403. Nevertheless, such request is never made on behalf of the client application and can only occur when the user tries to reach the server resource directly, via the provided API. That is why we do not have to handle this problem on the client side.

2.2 Blueprints

Blueprints can be created, viewed and modified until the day of the exam. All of these operations are quite obviously allowed only for users that hold the role of a teacher, and even those can manage only the blueprints that belong to the courses they teach.

2.2.1 Resources

Blueprints are exposed as two resources, each of them designed for different use. The first one represents a collection of blueprints and is accessible via `/api/blueprints`. It supports optional query parameters for filtering by subject, date and language. When no subject is specified, the result is automatically filtered using the list of subjects contained in the `User` service. If, on the other hand, a subject is given, it is tested against the same list and if no match is found, the server responds with a 403 code. Otherwise, the query is passed along.

In the end, the resource returns a collection of all the blueprints in the database that satisfy the inherent or supplied conditions. Anyhow, since the resource is used only for listings on the client side, it is unreasonable to request the actual content of the blueprints. Sending just the basic information will suffice and anything more than that would be a waste of bandwidth. To trim the response, the client can use the optional parameter `fields`. It is available to all the resources the API provides and when given, it prompts the server to return only the specified fields.

The other resource is used for operations dealing with a particular blueprint and can be found at URI `api/blueprint/{subject}/{date}/{language}`. Note that the variables are not optional and must be validated before any further processing, otherwise the server returns an error. The subject must be a valid subject code—starting with either `MI` or `BI` depending on the level of the subject, followed by a hyphen and ending with three word characters. The date must represent a string formatted as `YYYY-MM-DDThh:mm`, for example `2014-04-25T07:51`, and the language is validated against the standard *ISO 639-1* which allows only codes like en or `cs`.

The combination of these three variables is the only unique identifier of a blueprint in the application. There is a different type of key we might have used, which is the entry identifier assigned to every exam term in *KOS*, but because of the fact that an exam can be taken in different languages at once, it would not suffice. Even though such case is not very probable, it could still happen, and there is no reason for the application not to be as flexible as possible. It also helps retain a certain human-friendliness, as opposed to having to query the server API using generated identifiers.

2.2.2 Viewing blueprints

To view the blueprints, which only teachers are allowed to, the user must navigate to the state `Blueprints` which results in URL `/blueprints`. The items depicted are organized in stacks which reflect the logical distribution of blueprints among subjects. If only one subject is available, the user is presented directly with individual blueprints. Clear graphical distinction is made between what is considered stack and what is an individual "sheet" so that the user quickly recognizes which is which. Apart from using the browser history, the user can navigate the blueprints using a breadcrumbs navigation. This is possibly excessive in this case but makes more sense in different listings, for example when viewing exams where the structure depth can reach four levels. Also important is the fact that the current level is shared among all listings, which should help with the workflow. Anyhow, the listing is empty so far because we have yet to create our first blueprint.

2.2.3 New blueprint

To create a new blueprint, the user must enter the state `NewBlueprint` which is reflected in the URL as `/new`. She is then shown a listing similar to the one before, only this time with exam terms instead of blueprints. Terms are the particular dates on which an exam takes place—we can get those by reaching the *KOSapi* resource `/courses/{subject}/exams`, using each of the user's subjects for the variable, one after another. The client actually asks the application server resource `api/examterms` which delegates the request to the aforemen-

tioned endpoint. The result is then stripped of any term that is already fully occupied with blueprints in the database.

When a term is picked, the user is presented with a modal window requesting the language of the exam. This step is skipped if no choice is available or only one language is left. Afterwards, the user can finally proceed with the new blueprint.

The blueprint is initialized with one empty section that cannot be deleted and which is given a default name, unless changed. By using the controls, one can create another sections, move them up or down, or delete them along with their contents—as long as there is at least one section left. If needed, the user can also write a short introductory paragraph for the whole exam or for each of the sections.

When a new question is added to a section, the user must fill it up with content, which is divided into two parts—the body of the question and the answer. Both consist of a set of content blocks of different types that can be moved up or down the body and removed individually. This is achieved through controls pictured in the figure 2.3. When deleting a block—be it a chunk, a question or a whole section—the user is prompted if she really wishes to continue. If empty, the block is removed automatically.

The types of content that can be added to the question's body and answer are a little different, depending on the target. It would for example make no sense to add an external image to the answer or a list of options into the body. Otherwise, the choices are the same—the user can add a paragraph of text, a snippet of code, a list, or a canvas. In the case of a snippet, the user has to pick a programming language, which can be changed anytime later. If adding an external image, a link must be provided which is then validated by creating a JavaScript `Image` object having the URL for its source. If the object throws an error, it means the link is either broken or that the resource is not a valid image. Since downloading the image might take some time, it is necessary that the function—seen in the figure 2.4—returns a promise object, constructed via the service $q. Either way, the `Image` object is then destroyed, even if having passed the test—the image is already in the cache so there is no reason to keep it in the memory.

Figure 2.3: Creating a blueprint

```
function isImage(src) {
  var deferred = $q.defer(),
      image = new Image();

  image.addEventListener('error', function() {
    deferred.resolve(false);
  });
  image.addEventListener('load', function() {
    deferred.resolve(true);
  });

  image.src = src;
  image = null;

  return deferred.promise;
}
```

Figure 2.4: Validating an image

After creating the answer which usually consists of just one chunk of content, the user can fill it with a solution. This will help in the future evaluation. It is also possible to create a hint that takes a form of a pre-filled answer. When added, the hint has the same content as the solution. It is therefore necessary

to delete anything the user creating the blueprint does not wish the students to see.

When finished, the user can save the blueprint in the database. First, the blueprint is checked as to whether it meets the necessary conditions—it must contain at least one question and no question can be empty or without a defined answer. The blueprint is then saved into `localStorage`, in case something goes wrong. Only when successfully backed up, the process continues by converting chunks of the canvas type into SVG strings, and stripping snippets of any HTML—at least the kind that has been used for highlighting the code.

The blueprint is then finally passed to the server-side resource by means of `POST`. Before storing in the database, any SVG string inside the blueprint is optimized using the `SVGO` library—doing this can greatly reduce the size of the string. A blueprint stored in this way can later be modified, but only if it has not yet been used for an exam.

2.2.3.1 Cloning blueprint

While creating a blueprint, the user can browse the old blueprints via the `Blueprints` state like she would normally do. The difference is that she is now given an option to copy an already defined question into the new blueprint. It is also possible to copy an entire blueprint—any work already done will be replaced by an exact copy of the given blueprint, except the main identifiers like subject or date, which remain intact.

```
NewBlueprint.data.sections = JSON.parse(JSON.stringify($scope.
   blueprint.sections, function stripJunk(key, value) {
  if (/^[\$_]/.test(key)) {
    return undefined;
  }
  return value;
}));
```

Figure 2.5: Cloning a blueprint

One of the easiest and fastest way to clone an object in JavaScript is to

use the complementary functions `JSON.stringify()` and `JSON.parse()` in sequence. However, when cloning the blueprint, an extra caution must be taken not to copy properties such as `_id`, created while previously saving the object in the database. Otherwise, the new blueprint could not be stored. This can be done by employing the optional second parameter of `JSON.stringify()` which expects a function. The function can be used to reject a property depending on its name or value, and can be seen in the figure 2.5.

2.3 Exams

When referring to exams, it is important to note the difference between an exam term, as described before, and an exam in the sense of a particular student's exam.

2.3.1 Viewing exams

When viewing the exams via the state `Exams` located at `/exams`, the individual items represent available exam terms. The definition of what available means differs depending on the role of a user—a teacher sees any pending exam term that is listed in *KOS* and belongs to one of his subject. A student, on the other hand, is shown only the terms for which he applied in advance. Both lists can be obtained by using the already mentioned `api/examterms` resource.

2.3.2 Ongoing exams

By selecting an exam term, the client creates a new WebSocket connection which will be used by the server to handle communication among different users at the same exam. While this holds true for both students and teachers, what follows after is largely dependent on the user's role, it will therefore be explained separately.

2.3.2.1 Watching over exams

A teacher can see the exam—exactly as it will be shown to students—right from the beginning. Furthermore, he is given the option to temporarily show

the solution to any question. Until the moment the first student appears, he can also leave and therefore abort the exam. Whenever a student enters the exam, the teacher is sent a message containing all the necessary information. Since there can be more than one teacher watching over an exam, the message is actually sent to all of them. If any of these arrives late and identifies himself as a teacher, he is provided a list of students that are already present at the exam.

After making sure that all students at the exam are also physically present in the same room—which can be verified by the application itself by checking IP addresses of all the users and ensuring they all share the same subnetwork mask—any of the teachers can "hand out" the tests. That entails setting the exam duration. If necessary, the timer can be later adjusted. Nonetheless, when the time runs out, the exam is finished and all users are forced to leave. Teachers may also decide to collect the tests early, which has the same results.

2.3.2.2 Taking exams

A student can enter the exam only when a teacher is already present. Before allowed further he is also shown a modal citing a passage from the Ethical codex of CTU in Prague which he has to acknowledge in order to continue. While seemingly a superfluous formality, the reasoning behind this step is actually quite complex. The idea is based on the research by Dan Ariely, professor of psychology and behavioral economics, who led a series of experiments on cheating [21].

For a long time it was thought that people cheated by employing a rational cost/benefit analysis. When tempted to engage in an unethical behavior, they would weigh the chances of getting caught and the resulting punishment against the possible reward, and then act accordingly. However, experiments by Dr. Ariely and others have shown that far from being a deliberate, rational choice, dishonesty often results from psychological and environmental factors that people typically are not even aware of.

Ariely discovered this truth by constructing a set of experiments with group of students from the universities of MIT and Yale. The participants sat in a classroom-like setting and were given tests. To create an incentive, the
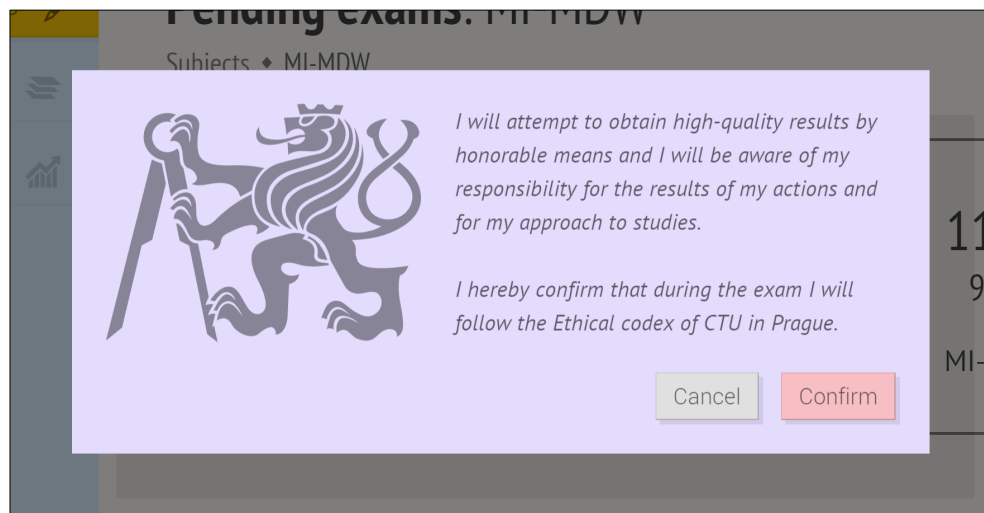
Figure 2.6: Acknowledging the Ethical codex

students were also given a certain amount of money for every right answer. Each of the experiments was then held under different conditions—making the cheating either less or more possible. In the end, Ariely found that given the chance, lots of people cheated—even if just a little.

Ariely then conducted another experiment, only this time the students had to sign a statement that said: "I understand that this experiment falls under the guidelines of the MIT/Yale honor code." The act of signing resulted in zero cheating, and this was true even though neither university actually has an honor code. As Ariely concluded from this line of his research, "recalling moral standards at the time of temptation can work wonders to decrease dishonest behavior and potentially prevent it altogether." Reminding the student that there is such a thing as the Ethical codex will hopefully have a similar effect.

Upon entering the exam, the menu is hidden and every attempt at navigating elsewhere manually is forbidden. The student is not allowed to see the test until a teacher "hands it out." When that happens, he is notified and told the time at which the exam ends. During the exam, the time left is visible in the form of a timer which calculates the value internally using the function `setInterval()`. Even though the single-threaded nature of JavaScript and the

inherent imprecision of its timing functions does not allow us to utilize the function directly to count the time, it can be used to periodically compare the current timestamp against the time of the exam's start. That way, we can obtain a fairly precise data that can be displayed to the user. To confuse the timer, the user might try to change the system time of his computer, which would work—but only locally. The end of the exam is determined exclusively by the timer running on the server. When that runs out, every user is forced to leave despite the time shown on their local timers.

The moment the user starts to answer the questions, his responses are sent to the resource `api/exam/{subject}/{date}/{language}/{uid}` using the `PUT` method and backed up in the database. This process is handled by a set of event listeners that are attached to each of the questions and which react to input. In order not to flood the server, the so-called debouncing is employed. It refers to a technique where a function is run only if a defined amount of time has passed since the last time it has been called.

A student can also "hand in" the test early, if he wishes to. The data is then prepared in the same way as in the process of creating a blueprint—canvases are transformed into SVG strings and snippets are stripped of any HTML. The result is then sent to the server and if successfully stored, the student is notified and redirected to the homepage, his navigation rights returned.

2.3.2.3 Prevention of cheating

While hopefully not an omnipresent factor, cheating still has to be taken into account—there are several ways the student could take advantage of the fact that the exam is taken on computer, even in the case of restricted network access. In order to prevent—and possibly discourage—the student from any attempt at cheating, two different techniques are used.

The first one employs the `blur` event of the `window` object which is fired whenever a page loses focus—either because the tabs in the browser were switched or because the browser itself lost focus in favor of some other application. When that happens, all teachers at the exam receives a notification pointing at the incriminated student.

Nonetheless, if applied by itself, a resourceful student still might think of a
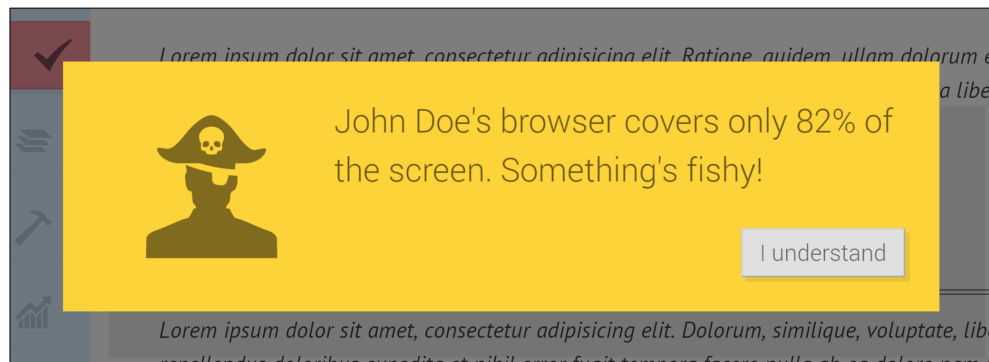
Figure 2.7: Notification of window resize

way to evade this countermeasure. If he prepared a file with notes in advance and resized the browser window so that the file can be seen—without having to switch between the applications—the first technique alone cannot stop him.

To handle the aforementioned situation, a different technique is utilized. Whenever a student tries to resize the browser, an event is fired. The client application on the student's computer then calculates the usage of the screen which is achieved by dividing the available viewport of the computer (`window.screen.availWidth` × `window.screen.availHeight`) by the dimensions of the browser window (`window.outerWidth` × `window.outerHeight`). If under 100 %, teachers are notified and shown a modal which informs them of the student in question and the percentage of screen covered by the browser. The same calculation also commences automatically at an exam start, in case a student resizes the screen in advance.

2.3.3 Evaluation

When finished, exams should be evaluated as soon as possible. If the logged-in user is a teacher and there is an exam that belongs to his subject and has not yet been evaluated, the item in the navigation menu that corresponds to the state `Evaluate` is highlighted in order to point out the fact.

After switching the state, the URL changes to `/evaluate` and the user is yet again presented with the already familiar listing. If necessary, the individual tests are grouped into exam terms and subjects. The only difference from
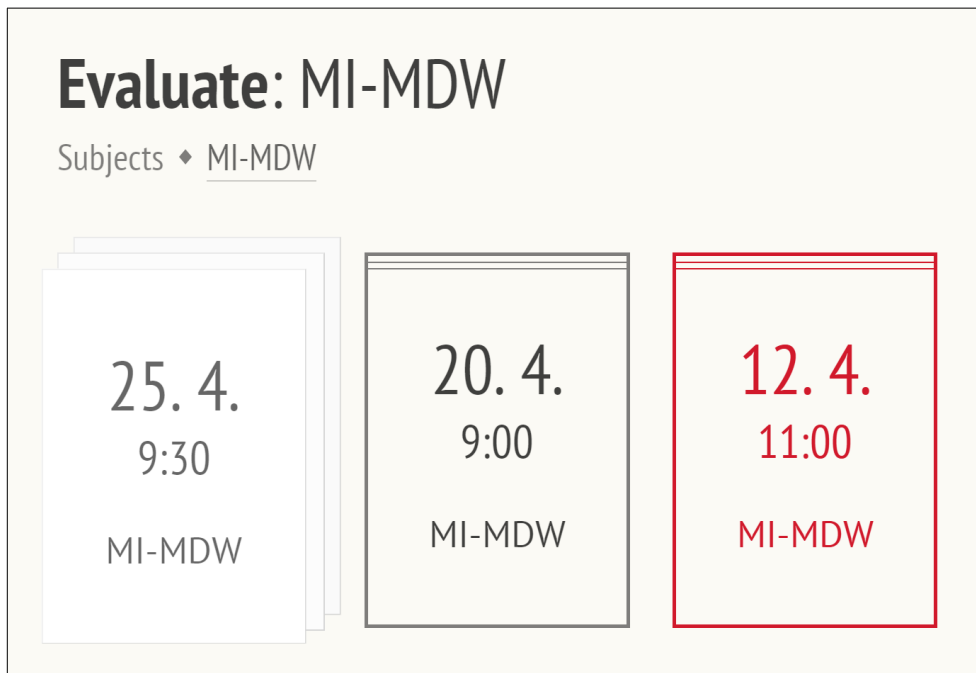
Figure 2.8: Example of listing

the other listings lies in the fact that both unevaluated tests and groups that contain at least one of these are highlighted. This can be seen in the figure 2.8. When the user selects one of the tests, he enters an evaluation mode which can either be canceled or finished by evaluating the test.

While evaluating, the user has the option to temporarily show the solution—if present—instead of the student's answer. Anyhow, each of the answers must be assigned a certain amount of points. These have an upper limit that was set while creating the blueprint for the exam, but no lower limit. The reason is to allow for special occasions where a teacher might consider giving a negative amount of points instead. Either way, when finished, the user can store the current evaluation and continue with other tests. If necessary, an exam test can also be re-evaluated which results in opening the finished evaluation, making additional changes, and then saving it as usual.

Conclusion

The application has successfully undergone the transformation from an abstract idea to a fully operational product. Nonetheless, one of the most important aspects of the development—testing the application with real users—still lies ahead. The absence of any usability testing could be also considered this work's greatest deficiency. Such testing would hopefully support the many ideas and opinions stated in the presented text but alas, time was not on our side.

Bibliography

[1] Node.js. [Online; accessed 6-May-2014]. Available from: `http://www.nodejs.org`

[2] Express. [Online; accessed 6-May-2014]. Available from: `http://expressjs.com`

[3] MongoDB. [Online; accessed 6-May-2014]. Available from: `http://www.mongodb.org`

[4] AngularJS. [Online; accessed 6-May-2014]. Available from: `http://angularjs.org`

[5] Mongoose. [Online; accessed 6-May-2014]. Available from: `http://mongoosejs.com`

[6] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*, chapter 5. University of California, Irvine, 2000. Available from: `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`

[7] Fette, I.; Melnikov, A. *The WebSocket Protocol [RFC6455]*. IETF. Available from: `http://tools.ietf.org/html/rfc6455`

[8] Socket.IO. [Online; accessed 6-May-2014]. Available from: `http://socket.io`

[9] Passport. [Online; accessed 6-May-2014]. Available from: `http://passportjs.org`

[10] Bower. [Online; accessed 6-May-2014]. Available from: `http://bower.io`

[11] Grunt. [Online; accessed 6-May-2014]. Available from: `http://gruntjs.com`

[12] Fabric.js. [Online; accessed 6-May-2014]. Available from: `http://fabricjs.com`

[13] Sass. [Online; accessed 6-May-2014]. Available from: `http://sass-lang.com`

[14] Harisov, V.; Belov, S. *BEM Methodology*. Yandex. Available from: `http://bem.info/method`

[15] Grigorik, I. *High Performance Browser Networking*. O'Reilly Media, 2013, ISBN 978-1-4493-4471-9.

[16] Rams, D. Ten Principles for Good Design. 1980. Available from: `http://www.vitsoe.com/eu/about/good-design`

[17] Korolkova, A.; Umpeleva, O.; Chaeva, I. Public Fonts for Minority Languages of Russia. 2011. Available from: `http://www.paratype.com/public`

[18] Brown, T. More Meaningful Typography. *A List Apart*, 2011. Available from: `http://alistapart.com/article/more-meaningful-typography`

[19] Revell, V.; Skene, D. Light-induced melatonin suppression in humans with polychromatic and monochromatic light. *Chronobiology International*, 2007. Available from: `http://www.ncbi.nlm.nih.gov/pubmed/18075803`

[20] Viola, A.; James, L.; Schlangen, L.; et al. Blue-enriched white light in the workplace improves self-reported alertness, performance and sleep quality. *Scandinavian Journal of Work, Environment & Health*, 2008. Available from: `http://www.ncbi.nlm.nih.gov/pubmed/18815716`

[21] Ariely, D. *The Honest Truth About Dishonesty*. Harper Perennial, 2013, ISBN 978-0-0621-8361-3.

Acronyms

API Application Programming Interface

CSS Cascading Style Sheets

DOM Document Object Model

DRY Don't Repeat Yourself

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object notation

LDAP Lightweight Directory Access Protocol

REST Representational State Transfer

SVG Scalable Vector Graphics

XML Extensible Markup Language

Contents of enclosed CD

```
 ┌ src .......................................... the directory of source codes
 │  └─ thesis ................the directory of LaTeX source codes of the thesis
 └─ text ........................................... the thesis text directory
    └─ thesis.pdf ............................. the thesis text in PDF format
```