

Web Applications, A.Y. 2020/2021
Master Degree in Computer Engineering
Master Degree in ICT for Internet and Multimedia

Homework 1 – Server-side Design and Development

Submission date: 23 April 2021

Last Name	First Name	Badge Number
Balzan	Pietro	1241795
Bettin	Manuel	1242413
Bullo	Marcello	1204533
Levorato	Nicola	1241689
Piva	Giovanni	2019282
Scicchitano	Wiliam	1241980

Objectives and main functionalities

The objective of the project, in continuation with the one developed during the database course, is to develop a web interface to manage requests among patients and doctors. In particular we want to implement a web application where both patients and doctors can interact easily. Our case study is about a group of doctors that work together with a pool of patients. This kind of medical office organization is very common in italy. From the doctor side, the web application provides different services such as: remote prescriptions, a patient's manager, an examinations manager and other interesting features dedicated to facilitate the cooperation with patients. From the patient side, the user can ask for prescriptions, exams, checkups and so on, without the need to go in person at the medical office.

Presentation Logic Layer

The project is divided into the following pages:

For any user:

- **Login page**: it allows the user (doctor, patient) to log in the web application and access to his/her protected area.
- **Registration page**: it allows a new patient to create his/her account. This page is accessible from any user since it is the first step in order to use the web application, but it is intended for patients only since doctors' management is delegated to the admin.

Exclusively for a doctor:

- **Homepage**: it is the welcome page of the patient where he/she can access the following pages.
- **Patients page**: it provides the doctor with a view of all the patients under his/her care. Whenever a doctor finds it necessary to take a new patient under his/her care, he/she can add such a patient to his/her own list.
- **Prescriptions page**: it allows doctors to see a list of all pending prescription requests and to accept or reject them. Doctors can see also the list of all past prescriptions (already accepted/rejected)
- **Profile page**: it allows a doctor to access its personal information as an overview of its own profile and change its password whenever it is necessary.

Exclusively for a patient:

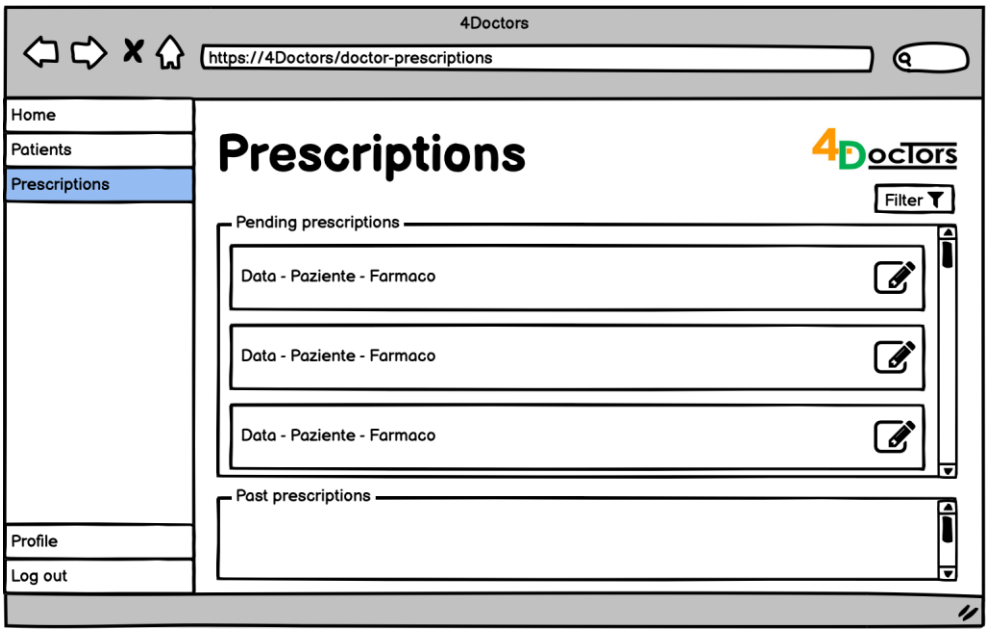
- **Homepage**: it is the welcome page of the patient where he/she can access the following facilities.
- **Prescriptions page**: it allows the patient to make prescription requests to a specific doctor for either exams and medicines. It also allows to display the list of prescriptions according to their status (pending, rejected, approved) and the list of the medicines.
- **Examinations page**: it allows the patients to make reservations for medical examinations from one of their doctors, choosing a specific date and time. Furthermore, the page shows two lists of the logged patient's examinations: one containing future appointments, and the other displaying all examinations that the patient took in the past.
- **Profile page**: it allows a patient to access its personal information as an overview of its own profile and change its password whenever it is necessary.

Exclusively for an admin:

- **Login page:** it allows the admin to log in the web application and access to his/her protected area. The admin has a different login URL located to /jsp/admin-login.jsp
- **Homepage:** it allows the admin to do the following operations:
    1. Review the list of all active doctors and for each one view his/her personal information or updating his/her status putting him/her inactive;
    2. Review the list of all patients and for each one view his/her personal information or removing him/her from the webapp;
    3. Register a new doctor in the webapp;
    4. Insert a new medicine in the database.

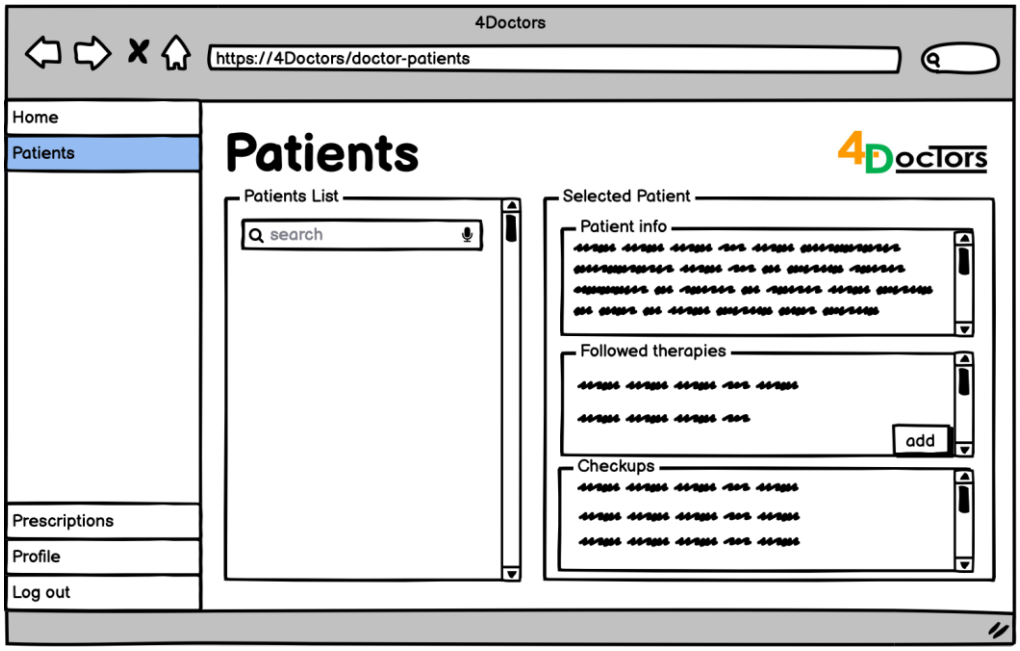
In the next paragraph, we will show and explain the interface mockups we made some of the pages of the webapp. The full set of UI mockups for all pages can be found in the project repository.

Doctor Prescriptions Page (Interface Mockup)



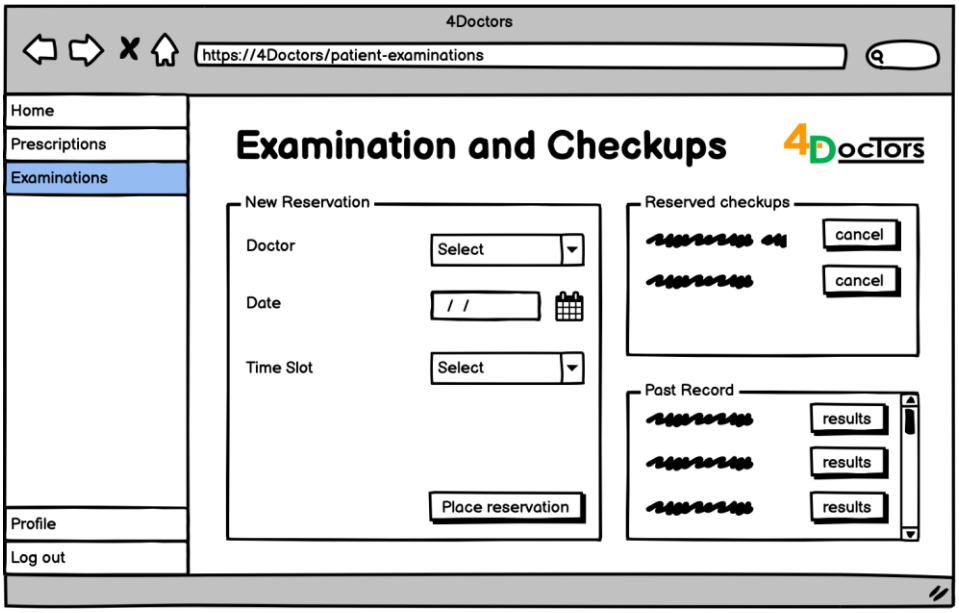
The page presents two scrollbar sections. The former contains a list of all pending prescription requests made by patients. Each request is a box with two main elements: on the left an overview of the prescription (patient, date…) and on the right a form where the doctor can accept or reject the corresponding request. The latter contains a list of all past prescriptions (prescription requests which have already been marked as approved or rejected). Each request has the same structure as before without the form part, in fact there are no operations to be performed since it is a past prescription request.

Doctor Patients Page (Interface Mockup)



From this page, a doctor can search for information about the patients under his/her care and he/she can also add patients to his/her *patients list* by adding their CF to the *patients list*. Furthermore, a doctor can delete a patient from his/her list but not from the list of other doctors, since a patient can be under the care of more than one doctor.
Future implementation could add more information about patients like a fast overview of his/her checkups and the display of his/her followed therapies.

Patient Examinations Page (Interface Mockup)

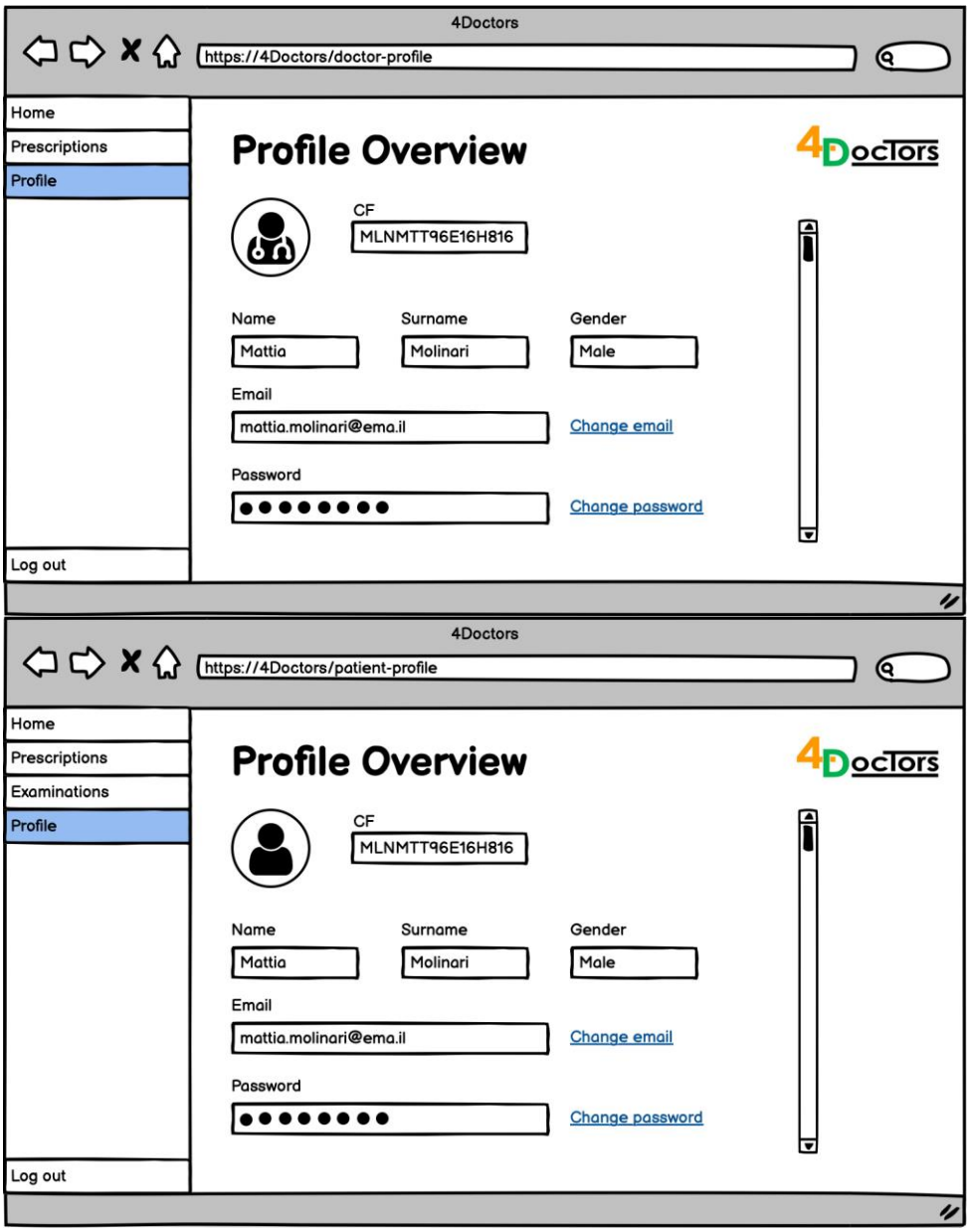


The page is split into three sections: a form that allows the patient to make a reservation for a medical examination, and two scrollbar sections that show both past and future examinations.

A new reservation is placed by selecting one of the doctors that are associated with the logged patient, as well as the preferred date and the time of the checkup. A patient will also be able to look at the results of past examinations, as well as to cancel the reservation for a future one.

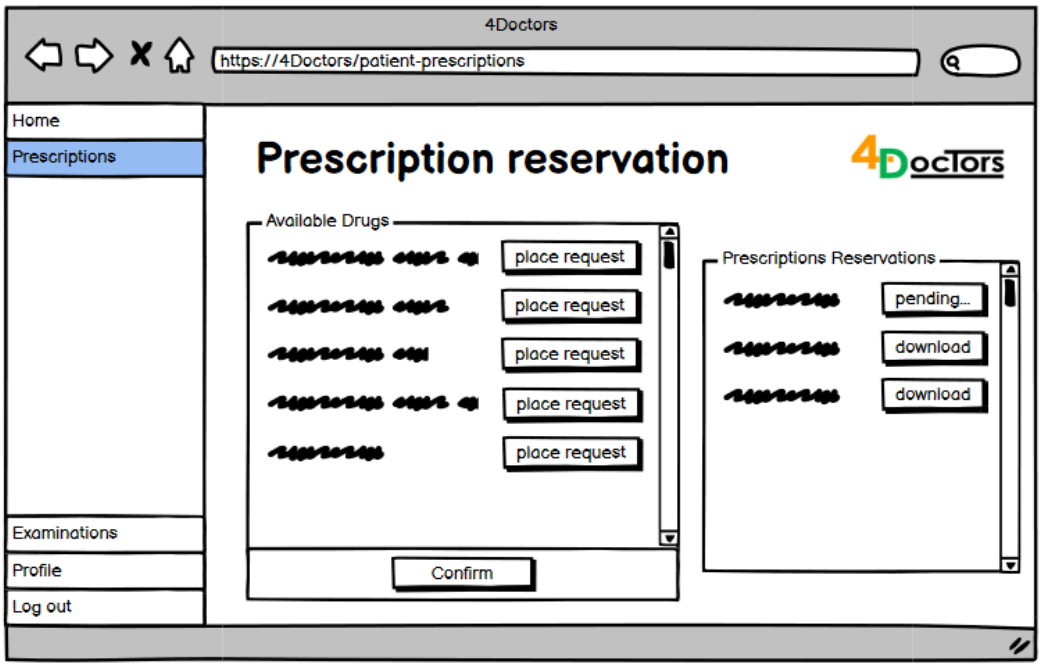
When the page is loaded, the two lists of past and future examinations are retrieved, and the “Doctor” selector is filled automatically with all the doctors that are associated with the logged patient, while the “Time slot” selector is filled with a standard set of bookable times (in the second homework, the list of selectable times should dynamically grey-out items in order to prevent the user to choose a time of the selected day in which other patients have already reserved checkups at the selected doctor).

Patient/Doctor Profile Page (Interface Mockup)



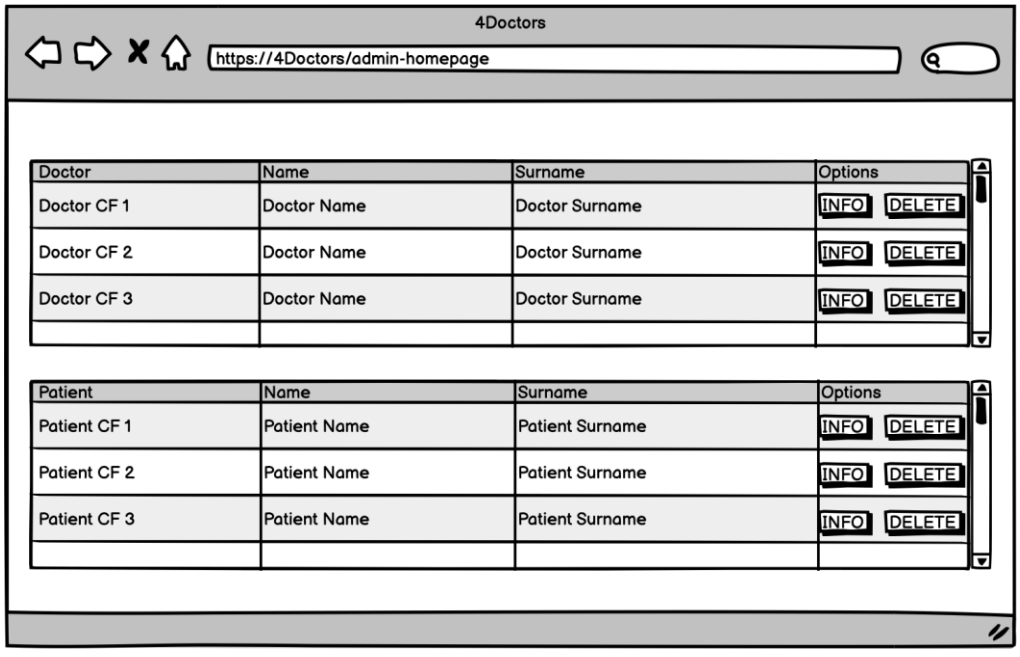
The patient/doctor profile page allows the user to access his/her personal information stored in the database. The page is intended to give to the user the possibility to edit his/her email to keep contacts updated and password, which is an important practice to adopt constantly to prevent security issues. For this reason the UI for this feature is thought to be as simple as possible. In fact, it shows the list of the personal information on the left with the change password and change email (as future implementation) buttons. Both the buttons redirect to a new page where a form allows to edit the intended information. Edits require a further password check.

Patient Prescription Reservation page (Interface Mockup)



The prescription reservation page allows patients to have the list of all their prescriptions, with the possibility to filter the request by the status of the prescription and see as result only the pending/approved/rejected ones. Moreover through this page the patient will be able to require a new prescription for medicine or exam. After a request, if the process ended correctly, the patient will see a new pending prescription in the list: it just must wait for a response from the doctor which will approve or reject the request.

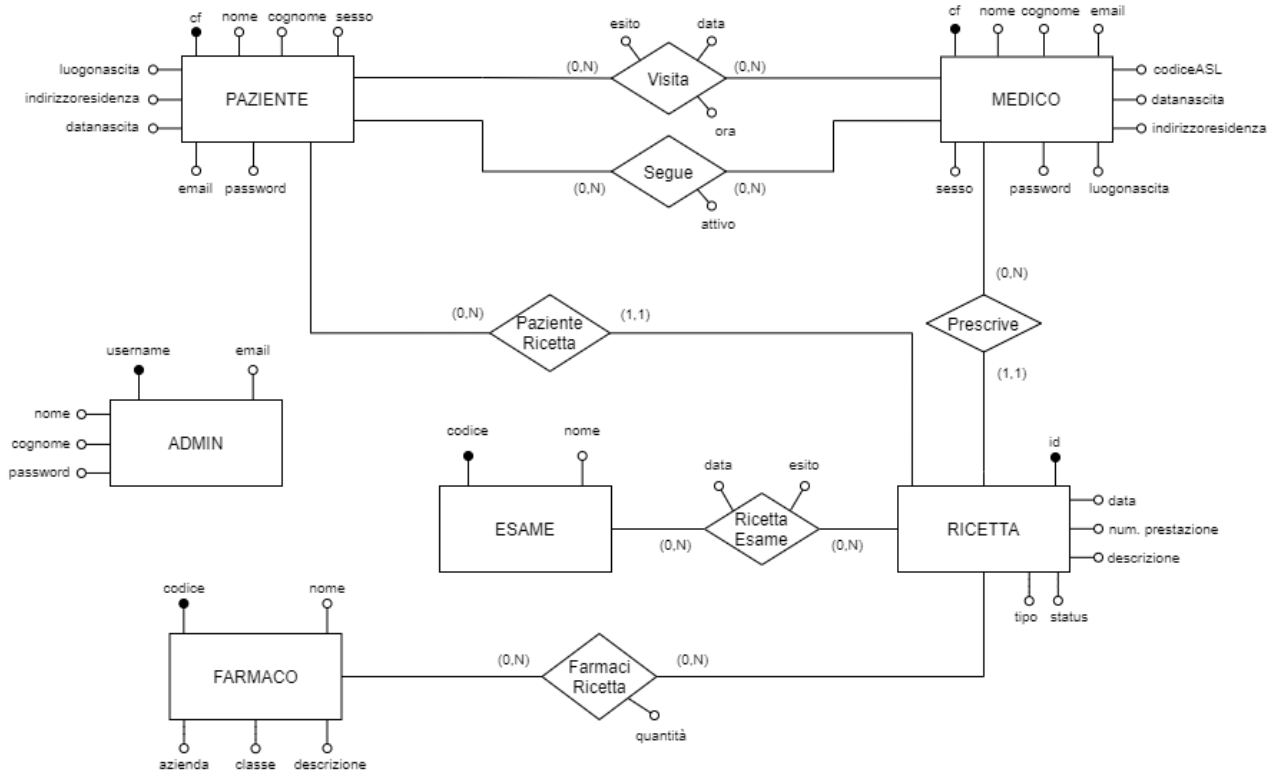
Admin Homepage (Interface Mockup)



The admin homepage allows admins to have the list of all the active doctors in the database that are the ones that follow at least one patient (there are at least a record in the relation Segue of the database between the doctor and a patient) and to have the list of all the patients registered in the webapp.
Also for each doctor or patient the admin can look at all his personal information clicking on the INFO button on the Options column and can also delete him clicking on the DELETE button.
Moreover through this page the admin will be able to add a new doctor or a new medicine (these two last mockups are inside the repository).

Data Logic Layer

Entity-Relationship Schema



- Paziente: contains all information relative to a patient. The fiscal code is used as primary key and together with the password is needed to login in the webapp. Then there is the email field (which is unique among all patients) needed to send communications from the webapp to the patient, for instance during the registration phase, when the password is changed and so on. Moreover we store all private information like name, surname, address, gender, birthdate and birthplace.
- Medico: similarly to patient, the doctor table stores all main information about doctors. Fiscal code and password allow access to the doctor's area in the webapp. The codiceASL is relative to the city where the doctor is placed, while other fields concern all other private information.
- Ricetta: each prescription is uniquely identified by an id of UUID type. The main features are the type attribute, which points out whether the prescription is requested for an exam or a medicine and the status which distinguish among approved, rejected or still pending requests. Even more, there are further recorded information such as the date when the request has been made and a small description of it.

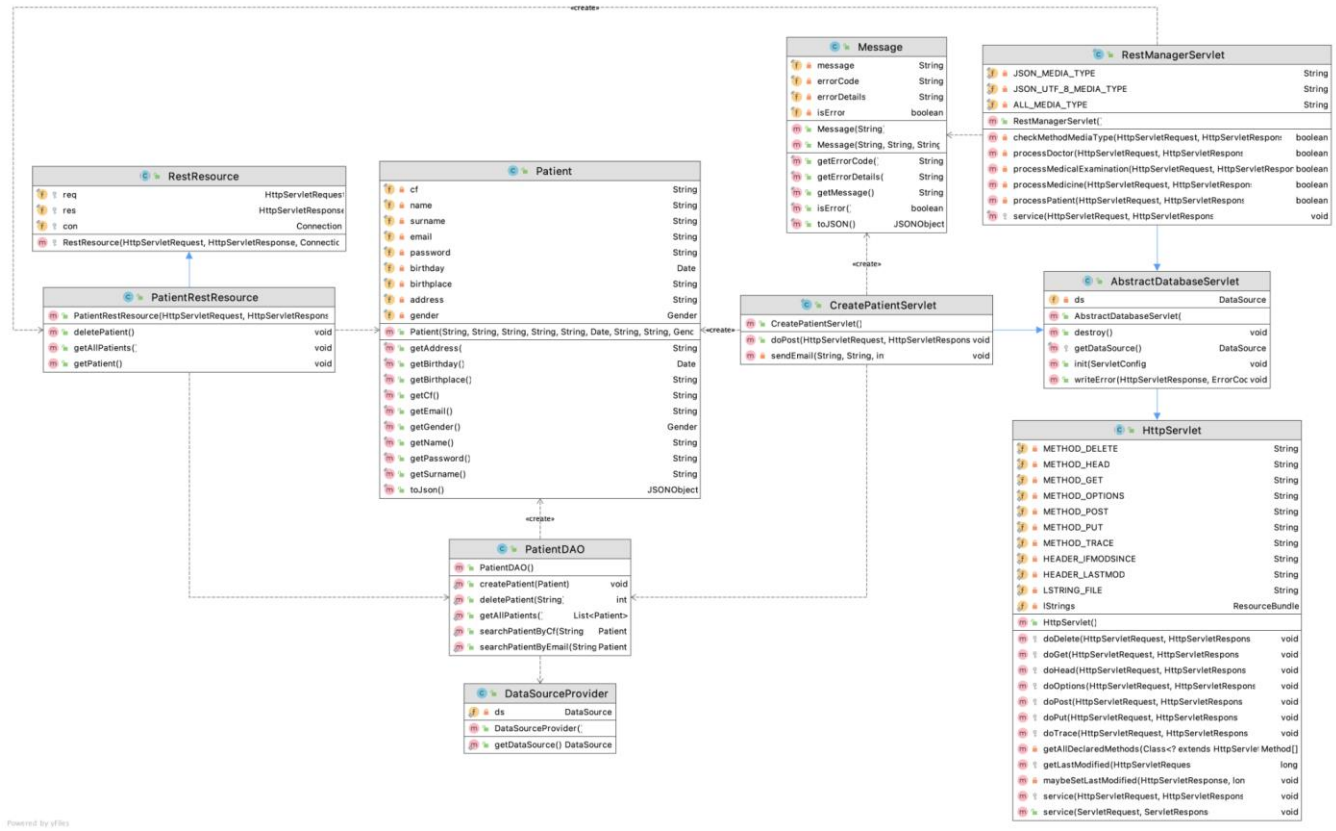
- Esame: in the database we stored a lot of different exams, each with a unique code which is the primary key. The name is a “user friendly” sentence which allows patients to request a prescription for whatever exam they need. If the request is approved by a doctor, the date and the result of that exam will be registered in the RicettaEsame table (null values are used otherwise).
- Farmaco: medicines are stored in this table. Each medical has a code as unique identifier and some advanced information such as the name, a description, the producer. Patients have access to the whole list of medicines stored in the database and they can request a prescription for one of them. After that, a doctor can decide whether to approve or reject the request, either because he/she considers it appropriate or not for that patient.
- Admin: this table represents the admin users, which are the only users able to delete/insert new doctors and new medicines in the database and to look at the list of all patients and active doctors stored in the database. The username is used as primary key and together with the password is needed to login in the protected admin area. Moreover we store all private information like name, surname and email (which is unique among all admins).

Other Information

Most of the queries can be directly made through the webapp interface. In the patient area, patients can look at their own profile data and request for medicines or exams. On the other hand, in the doctor area, doctors can look to the profile overview, see the whole list of patients and manage their requests. All the queries concern SELECT, INSERT and UPDATE operations. Nobody is allowed to make a DELETE operation in any tables (except for the admin user. This is due to a complete tracking policy. For instance, rejected prescriptions are still stored in the db even if they seem not useful anymore).
We create many custom enumerations to capture the correct meaning of some attribute. For example, the status of a prescription can be only ‘PENDING’, ‘APPROVED’ or’ REJECTED’ while its type can be either ‘FARMACO’ or ‘ESAME’.

Business Logic Layer

Class Diagram - Patient

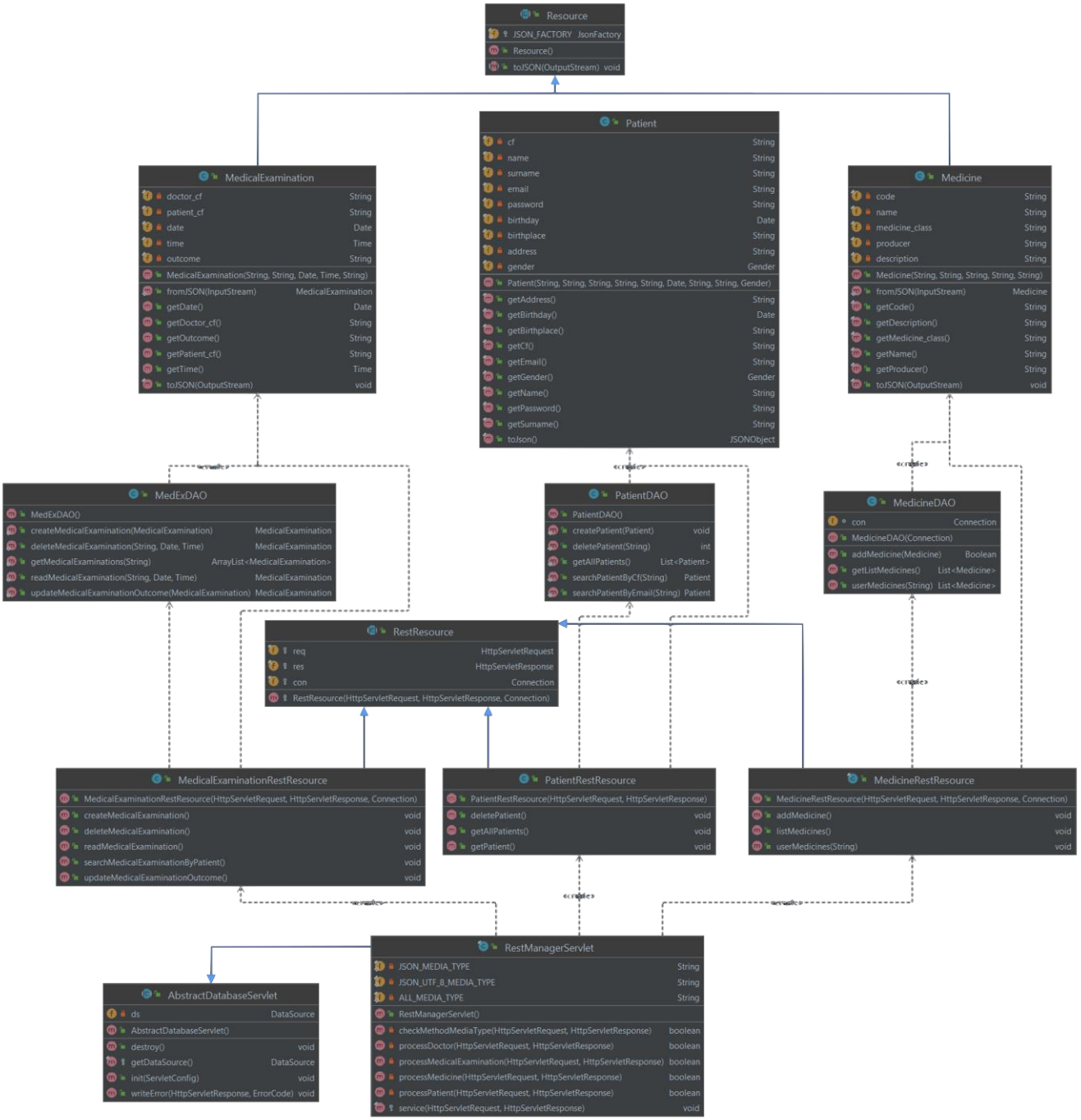


The class diagram contains (some of) the classes used to handle the *Patient* class.
It is possible to notice that to create a patient and store it in the database we use the `CreatePatientServlet`, a subclass of `AbstarctDatabaseServlet`, which implements:

- `doPost` method to handle the POST request sent when a new patient succeed in fulfilling the registration form and get access to his/her own protected area;
- `sendEmail` method to send a confirmation code necessary during the registration procedure.

The creation of a patient requires a connection to the database and, consequently, the execution of a specific query. Both the connection and query execution are handled through the `PatientDAO` class which uses `DataSourceProvider` class to get the connection from the database and implements the static method `PatientDAO.createPatient` to execute the relative query.
Concerning the REST functionalities, in order to have a more detailed overview, see the class diagram below.

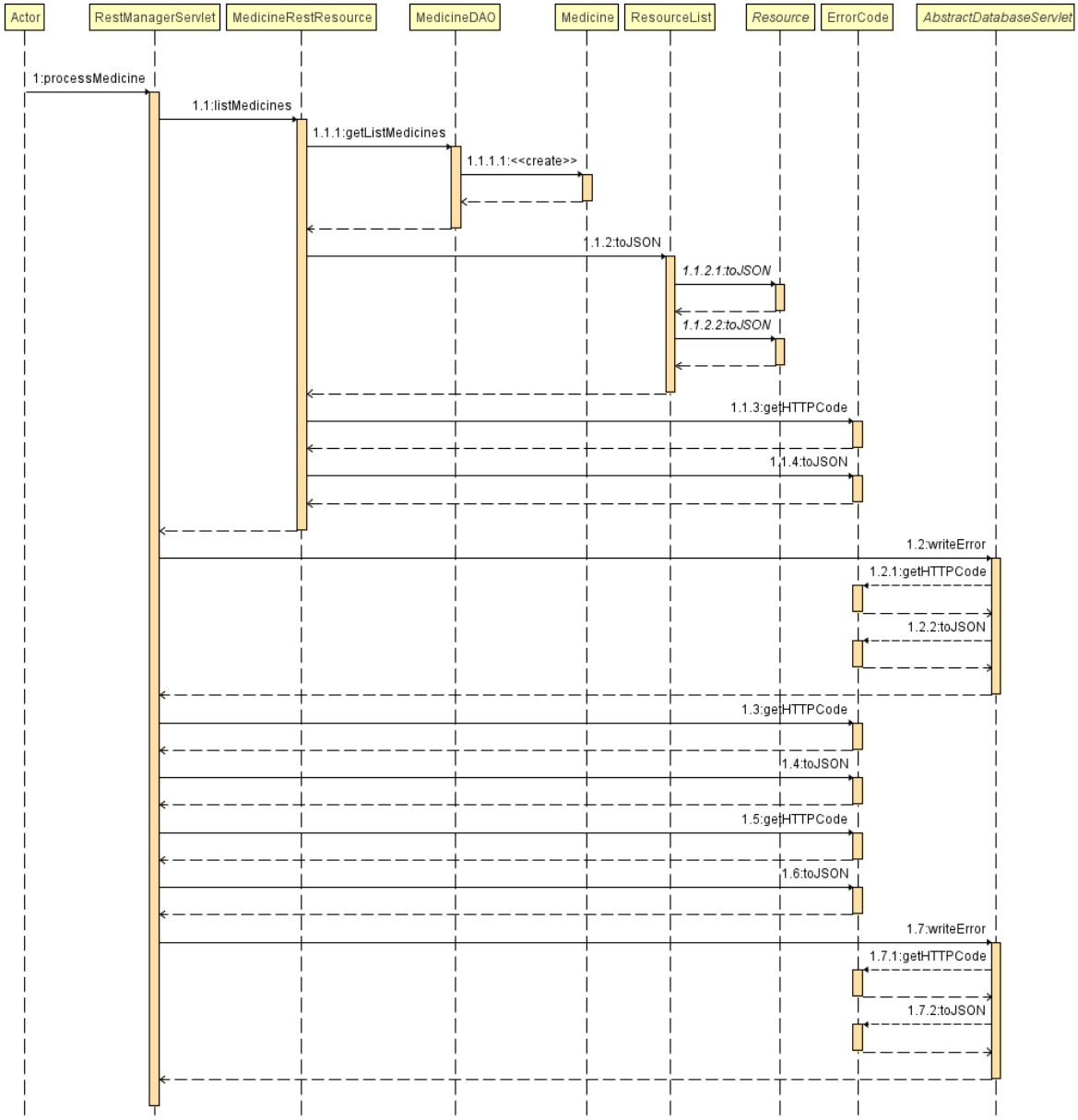
Class Diagram - REST implementation



The above UML diagram shows some of the classes involved in the REST paradigm implementation of the first homework and their interaction (the description starts with the bottom items of the UML and proceeds upwards). The three involved resources are the `MedicalExamination`, `Medicine`, and `Patient` resources, which all have quite a similar implementation and functioning.
The servlet that implements the required behaviours to handle the shown resources (as well as others) is a traditional servlet based on the REST paradigm: the `RestManagerServlet`, which extends the `AbstractDatabaseServlet`. The servlet processes all rest calls and determines which resource is being called in its `service` method by calling, one by one, the processor methods of each supported resource: `processMedicalExamination`, `processPatient`, `processMedicine`, as well as others for the not shown resources. It also checks if the request method and MIME media type are

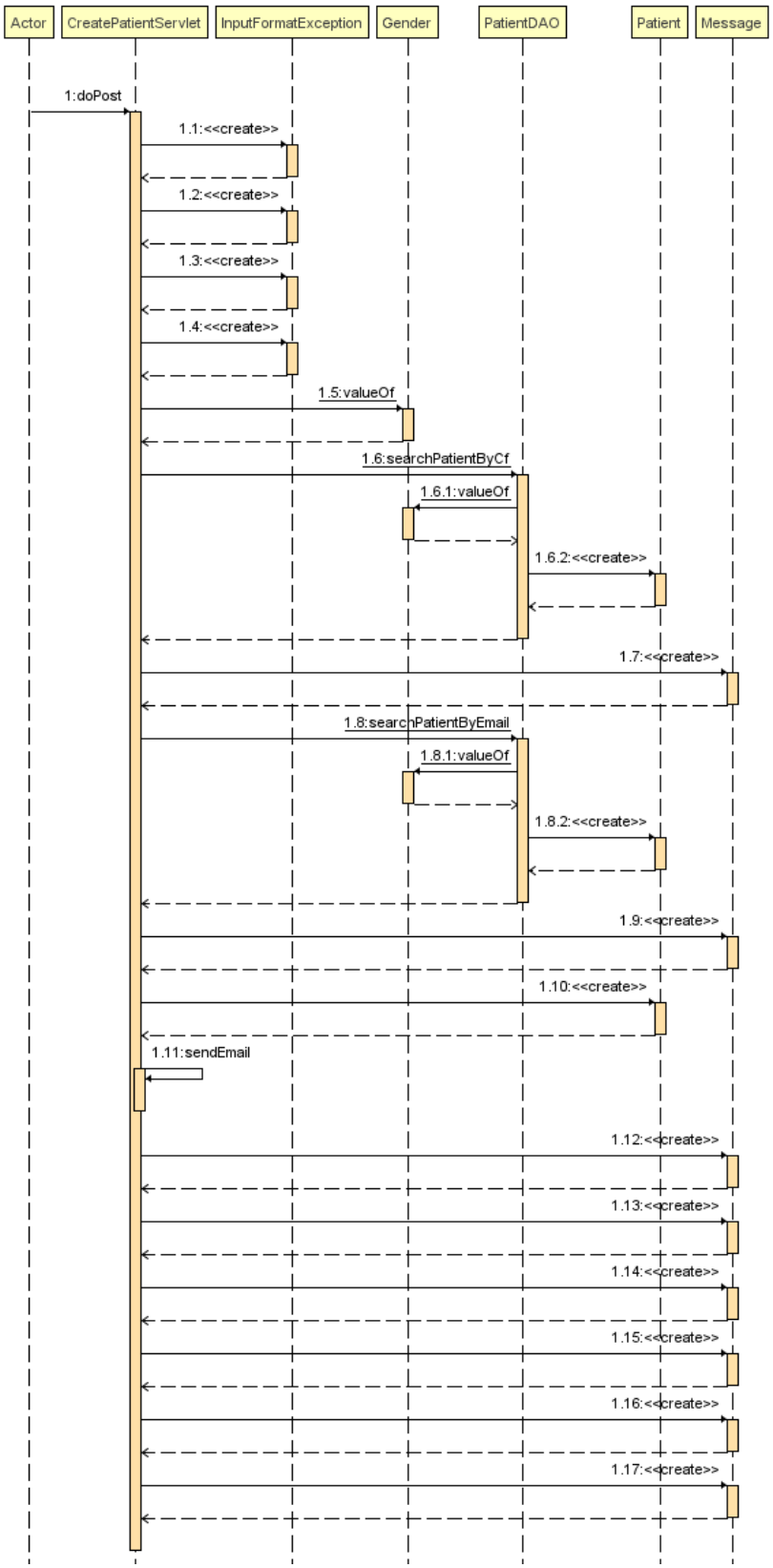
allowed or not by calling `CheckMethodMediaType`, and shuts the request down in case of a negative outcome. If any of the resources are identified, the URI is parsed further and, once the request has been fully understood, the appropriate `RestResource` class method is called to start the interaction with the database and fetch, delete or update information. In order to achieve this, these methods perform calls to the relative DAO classes (for example, as you can see from the above image, the `MedicineRestResource` will use the corresponding `MedicineDAO` class to interact with the db) and, depending on the requirements of the request, usually a single or a list of resources is returned to the client as a JSON. Each DAO class implements all useful methods and operations to handle the relative resource in the database: `MedicineDAO` is used by calling the class constructor first, while `MedExDAO and DoctorDAO` use static methods, but all three use the `Datasource` singleton object to gain access to the database.

Sequence Diagram



Here reported the sequence diagram for the listMedicines operation: the user executes a GET request to the RestManagerServlet specifying the URI medicine/. The servlet instantiates an object from

MedicineRestResource, passing to the constructor: the request, the response and the connection to the datasource. On the object so far instantiated the servlet calls the method listMedicine, which one accesses the database by means instantiating a MedicineDAO object and invoking on it a method called getListMedicines. Once the list of medicines is returned it will be written in a JSON format and send back to the client the response in JSON format.



Here reported the sequence diagram for the createPatient operation: the user registers to the webapp through the patient_registration page doing a POST operation. The operation calls the doPost method of the *CreatePatientServlet*. If there are any errors in the data inserted by the user in the form for the registration a *InputFormatException* is created and a *Message* object containing the error to visualize is sent back to the jsp page. Before adding the patient in the database, the servlet checks if the patient is already stored using the methods *searchPatientByCf* and *searchPatientByEmail* of PatientDAO. Then an email for the verification is sent to the user's email using the *sendEmail* method. Finally the process of adding the patient into the database is delegated to the VerificationServlet.

REST API Summary

The rest API is studied to experiment with multiple different approaches to the REST paradigm.
The operations associated with the Medical Examinations are developed in the traditional way, with all the parameters needed to satisfy the request already being present in the URI.

URI	Method	Description
/rest/medicalExamination	POST	Creates a new medical examination with the parameters specified by a Json.
/rest/medicalExamination/{doctor_cf}/{date}/{time}	GET	Reads a medical examination from the database. The three fields are used to specify the primary key of the examination.
/rest/medicalExamination/{doctor_cf}/{date}/{time}	PUT	Updates the outcome of an existing medical examination with the parameters specified by a Json. Only the "outcome" field must differ in order for the operation to successfully complete.
/rest/medicalExamination/{doctor_cf}/{date}/{time}	DELETE	Delete an examination from the database. The three fields are used to specify the primary key of the examination to delete.
/rest/medicalExamination/patient/{patient_cf}	GET	Searches and returns a list of all medical examinations of a patient, based on the patient's fiscal code {patient_cf}.
/rest/medicine	GET	List all the medicines available in the database
/rest/medicine	POST	Create a new medicine in the database
/rest/medicine/{patient_cf}	GET	List all the medicines prescribed to the patient identified by {patient_cf}
/rest/patient/list	GET	List all the patients available in the database.

/rest/patient/{patient_cf}	GET	Get all the information about a patient stored in the database identified by {patient_cf}.
/rest/patient/{patient_cf}	DELETE	Delete a patient from the database identified by {patient_cf}.
/rest/doctor/list	GET	List all the active doctors (the ones that follow at least one patient) available in the database.
/rest/doctor/{doctor_cf}	GET	Get all the information about a doctor stored in the database identified by {doctor_cf}.
/rest/doctor	POST	Create a new doctor in the database.
/rest/doctor/{doctor_cf}	PUT	Update the status of the doctor identified by {doctor_cf} making it no more active in the database. The operation is possible only if the doctor is following at least one patient.
/rest/doctor/{doctor_cf}	DELETE	Delete a doctor from the database identified by {doctor_cf}.

REST API Error Codes

Now we list all the error codes used by the rest application. Internal errors, which correspond to crashes, servlet exceptions, or problems with the input/output streams are identified with the Error Code E5G12.

Error Code	HTTP Status Code	Description
E4D1	NOT_FOUND	Doctor not found
E5D2	INTERNAL_SERVER_ERROR	Failed to create a doctor
E5D3	CONFLICT	Cannot create the doctor, it already exists
E5D4	INTERNAL_SERVER_ERROR	Cannot update doctor status because is already inactive
E4D5	METHOD_NOT_ALLOWED	Unsupported operation for URI /doctor.
E4P1	NOT_FOUND	Patient not found
E5P2	INTERNAL_SERVER_ERROR	Failed to create patient
E5P3	CONFLICT	Cannot create the patient: it already exists.
E4P4	METHOD_NOT_ALLOWED	Unsupported operation for URI /patient.
E5M1	INTERNAL_SERVER_ERROR	Failed to create medicine
E5M2	INTERNAL_SERVER_ERROR	Failed to search medicine
E5M3	CONFLICT	Cannot create the medicine: it already exists
E4M4	METHOD_NOT_ALLOWED	Unsupported operation for URI /medicine
E4M5	BAD_REQUEST	Cannot serve the request: invalid cf
E5V1	INTERNAL_SERVER_ERROR	Cannot create the medical examination

E5V2	BAD_REQUEST	Cannot create the examination: date or time are not valid.
E5V3	INTERNAL_SERVER_ERROR	Failed to search medical examination
E5V4	INTERNAL_SERVER_ERROR	Failed to search medical examination: error while parsing URI
E5V5	CONFLICT	Cannot create the medical examination: it already exists.
E4V6	METHOD_NOT_ALLOWED	Unsupported operation for URI /medicalExamination
E4G1	BAD_REQUEST	The data is in the wrong format
E4G2	BAD_REQUEST	The input json is in the wrong format
E4G3	BAD_REQUEST	Requested an unknown operation
E4G4	METHOD_NOT_ALLOWED	Method requested not implemented
E4G5	BAD_REQUEST	Media type not specified
E4G6	NOT_ACCEPTABLE	Unsupported media type. Resources are represented only in application/json
E5G12	INTERNAL_SERVER_ERROR	Internal Server Error

REST API Details

Here we show a wide range of functionalities relative to the REST resources present in the project. Most of the methods and operations shown here require a JSON in the input or produce it as the output or the operation, or both.

Medical Examination
The following endpoint allows access to a specific medical examination, and to perform operations such as read, update or deletion of the resource.

- URL
  medicalExamination/

- Method
  GET | PUT | DELETE

- URL Params
  {doctor_cf}/{date}/{time}

  Required:
  - `doctor_cf = {string}`
    The fiscal code of the doctor performing the checkup.
  - `date = {string}`
    The date on which the checkup was reserved, formatted as yyyy-MM-dd.
  - `time = {string}`
    The time on which the checkup was reserved, formatted as hh:mm.

As an additional requirement, only times present in the BookingTime class will be successfully accepted.
(minutes of the time can only be :00 and :30, and the hours can only be values between 09-12 and 14-18).

- Data Params
  No data needs to be passed for the GET and DELETE methods, while the PUT method requires to send the JSON containing the info of the medical examination with the updated outcome.

  Required:
  - `doctor_cf = {string}`
    The fiscal code of the doctor performing the checkup.
  - `patient_cf = {string}`
    The fiscal code of the patient requiring the checkup.
  - `date = {string}`
    The date on which the checkup was reserved, formatted as yyyy-MM-dd.
  - `time = {string}`
    The time on which the checkup was reserved, formatted as hh:mm.
    The same restrictions on the possible values of the time still apply.
  - `outcome = {string}`
    The updated outcome of the examination. This way, a Doctor can update the outcome after performing the checkup on the patient.

- Success Response
  Upon success, the server returns a JSON of the read/updated/deleted medical examination.

  Code: 200
  Content:

```
{
 "medicalExamination":
      {
            "doctor_cf":"F345MED1K0283759",
            "patient_cf":"GVNPVAG4RE44S3D9",
            "date":"2022-04-23",
            "time":"16:30",
            "outcome":"esito aggiornato."
      }
}
```

- Error Response

  Code: 500 INTERNAL_SERVER_ERROR
  Content:

```
{     "error": {
            "code": "E5V3",
            "message": "Failed to search Medical Examination."
      }
}
```

  When: the medical examination specified by the URI was not present in the database.

  Code: 500 INTERNAL_SERVER_ERROR

Content:

```
{     "error": {
            "code": "E5V4",
            "message": "Failed to search Medical Examination: error
                        while parsing URI."
      }
}
```

When: the URI didn't contain the required number of fields after "medicalExamination/".

Code: 500 INTERNAL_SERVER_ERROR
Content:

```
{     "error": {
            "code": "E5G12",
            "message": "internal server error."
      }
}
```

When: a generic unspecified internal error happens in the server.

Medicine Prescription

The following endpoint allows access to a specific medicine prescription, and to perform operations such as read the entire list of all available medicines in the dataset or add a new resource.

- URL
  medicine/

- Method
  GET | POST

- URL Params
  No url params required

- Data Params
  No data needs to be passed for the GET method, while the POST method requires to send the JSON containing the info about the medicine to be added in the dataset.

  Required:
  - `code = {string}`
    The code which uniquely identifies the medicine.
  - `name = {string}`
    The name of the medicine.
  - `medicine_class = {string}`
    The class that the medicine belongs to. It could be "SOP", "OTC" or "ETICI".
  - `producer = {string}`
    It's the pharmaceutical company which produces the medicine.
  - `description= {string}`
    A brief description of the medicine, usually the active principle.

- Success Response
  Upon success, for the GET method, the server returns a JSON of the read list of medicines. For what about the POST method the server responds with the HTTP status code: 201 Created.

Code: 200
Content:

```
{
  "resourceList":
                [{"medicine":{
                                "code":"AG3T46E9D0",
                                "name":"Brufen",
                                "medicine_class":"ETICI",
                                "producer":"Mylan",
                                "description":"Corticosteroidi"}},
                {"medicine":{
                                "code":"234DF21345",
                                "name":"Broncovaleas",
                                "medicine_class":"SOP",
                                "producer":"Valeant",
                                "description":"Salbutamol Solfato"}}]
}
```

- Error Response

Code: 405 METHOD_NOT_ALLOWED
Content:

```
{     "error": {
                "code" : "E4M4",
                "message" : "Unsupported operation for URI /medicine."
      }
}
```

When: the method on the request is different from POST or GET .

Code: 400 BAD_REQUEST
Content:

```
{     "error": {
                "code" : "E4M5",
                "message"  :  "Cannot  serve  the  request:  invalid
          parameter."
      }
}
```

When: the URI contains something which is not a possible fiscal code after “medicine/”.

Code: 500 INTERNAL_SERVER_ERROR
Content:

```
{     "error": {
                "code" : "E5G12",
                "message" : "internal server error."
      }
}
```

When: a generic unspecified internal error happens in the server.

Code: 500 INTERNAL_SERVER_ERROR
Content:

```
{     "error": {
                "code" : "E5M2",
```

```
                    "message" : "Failed to search medicine"
        }
    }
```
When: an error occurs while the server access the dataset

Code: 500 INTERNAL_SERVER_ERROR
Content:
```
    {     "error": {
                    "code" : "E5M1",
                    "message" : "Failed to create medicine"
        }
    }
```
When: an error occurs during the insertion of the new medicine in the dataset.

Code: 409 CONFLICT
Content:
```
    {     "error": {
                    "code" : "E5M3",
                    "message" : "Cannot create the medicine: it already
              exists."
        }
    }
```
When: we try to insert a new medicine with a code already present in the dataset.

Patient
The following endpoint allows access to a specific patient, and to perform operations such as read the entire list of all patients in the dataset.

- URL
  patient/list

- Method
  GET

- URL Params
  No url params required

- Success Response
  Upon success, for the GET method, the server returns a JSON of the read list of patients.

  Code: 200
  Content:
```
          {
             "data": {
                "patients-list": [
                   {
                      "birthday": "1996-05-16",
                      "password": "098f6bcd4621d373cade4e832627b4f6",
                      "cf": "MLNMTT96E16H816G",
                      "address": "Via zef,9,Forgaria Nel Friuli(UD),33030",
                      "gender": "M",
```

```
            "birthplace": "San Daniele Del Fr.",
            "surname": "molinaro",
            "name": "mattia",
            "email": "email1"
        },
        {
            "birthday": "1997-01-23",
            "password": "098f6bcd4621d373cade4e832627b4f6",
            "cf": "GVNPVAG4RE44S3D9",
            "address": "Via Cesare Battisti, 124 (PD), 35121",
            "gender": "M",
            "birthplace": "Padova",
            "surname": "piva",
            "name": "giovanni",
            "email": "email4"
        },
    ]
  }
}
```


- Error Response

  Code: 405 METHOD_NOT_ALLOWED
  Content:


```
{
    "error": {
        "code": "E4P4",
        "message": "Unsupported operation for URI /patient."
    }
}
```


  When: the method on the request is different from GET .

  Code: 500 INTERNAL_SERVER_ERROR
  Content:


```
{
    "error": {
        "code": "E5G12",
        "message": "Internal server error."
    }
}
```


  When: a generic unspecified internal error happens in the server.

Doctor
The following endpoint allows access to a specific doctor and to read all his specific information.

- URL
  doctor/{doctor_cf}

- Method
  GET

- URL Params
  doctor_cf = {string} : The cf of the doctor to search in the database.

- Success Response
  Upon success, for the GET method, the server returns a JSON with all information about the specific doctor requested.

  Operation: doctor/SON01MED1C0G1UR0
  Code: 200
  Content:

```
{
   "data": {
      "doctor": {
         "birthday": "1965-06-11",
         "cf": "SON01MED1C0G1UR0",
         "address": "Via dei medici, 118 (VE), 34761",
         "birthplace": "Venezia",
         "gender": "M",
         "surname": "rossi",
         "name": "andrea",
         "aslcode": "A34VE",
         "email": "andrea.rossi@gmail.com"
      }
   }
}
```

- Error Response

  Code: 405 METHOD_NOT_ALLOWED
  Content:

```
{
    "error": {
        "code": "E4D5",
        "message": "Unsupported operation for URI /doctor."
    }
}
```

  When: the method on the request is different from GET, PUT or DELETE.

  Code: 500 INTERNAL_SERVER_ERROR
  Content:

```
{
    "error": {
        "code": "E5G12",
        "message": "Internal server error."
    }
}
```

  When: a generic unspecified internal error happens in the server.

Code: 500 INTERNAL_SERVER_ERROR
Content:

```
{
    "error": {
        "code": "E4D1",
        "message": "Doctor not found"
    }
}
```

When: an error occurs during the searching of a doctor in the dataset.