RESTful Services

iOS Praktikum WS 14/15 – Introduction Course

Stephan Rabanser
Department of Computer Science
Technische Universität München
rabanser@cs.tum.edu

November 20, 2014

Contents

List of Figures

List of Tables

1 Introduction

Representational State Transfer (REST) is an architectural style for distributed systems and web-services. REST is well suited for high-level client-server communication and is extensively used in many iOS applications.

1.1 Motivation

REST is a cross-platform, common architectural standard for network communication, especially on the internet. The World Wide Web, for example, represents the largest implementation of a system conforming to the REST architectural style. Fortunately, REST is quite simple, well understood by the developer community and widely adopted. It is also easy to consume for thin clients, which makes it especially suitable for iOS applications.

1.2 Example iOS applications

Almost every iOS app which relies on client-server communication interacts with a server conforming to REST.

The following iOS applications make extensive use of RESTful communication:

- Instagram[1]: The Instagram app talks to a RESTful API to post and retrieve photos.
- Dropbox[2]: The Dropbox app retrieves the list of all files you currently have in your Dropbox by interacting with a RESTful server.
- Twitter[3]: The Twitter app communicates with a RESTful service to get the current home timeline, mentions and direct messages, but also to post new tweets.

2 iOS Intro architecture overview

First, it is important to become familiar with the iOS Intro application architecture, since we are going to rely on this model for the rest of the tutorial.

The server exposes various API endpoints through its Jersey[4] REST interface. The latter is connected to a MySQL instance running on the same server. We are using JPA in order to keep data in sync between the REST interface and the database.

On the iOS client, we are using AFNetworking[5], an external networking library, to connect to the interfaces exposed by the server. In most cases you would also want to use CoreData as a persistence context and manager for the underlying SQLite database.

A complete REST call from request to response would look as follows: The iOS client sends a REST request to the server by using AFNetworking's built in functions. When the request hits the server, it automatically passes it to the Jersey REST library, which is responsible for parsing the request. In many cases, the request asks for some kind of data which is stored in the database. In such cases, we need to query the database, which is done by using JPA; there are no SQL queries involved here. The resulting data then gets passed up again to Jersey, gets formatted in a proper way and is eventually returned to the client. At this point, the client can store the received data inside of its local database.

[1] `http://instagram.com/developer/endpoints/`
[2] `https://www.dropbox.com/developers/core/docs`
[3] `https://dev.twitter.com/rest`
[4] `https://jersey.java.net`
[5] `http://afnetworking.com`

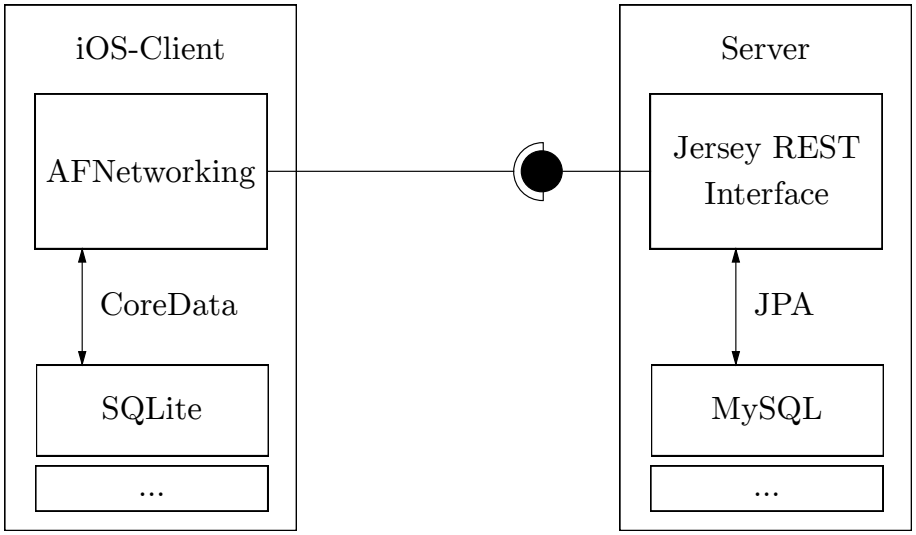

Figure 1: iOS Intro architecture overview

3 REST Requests

Whenever a client wants to perform some kind of action to a specific set of data on the server, it has to send a dedicated request to the server. The most important components of such a request are listed in Table 1. In this tutorial, we are mostly going to concentrate on the request line.

HTTP Request	Request line
	Headers
	Message Body Data

Table 1: HTTP Request

The REST request line – also called REST call – is build up by two different parts: the REST command and the URI.

```
REST request = REST command + URI
```

3.1 REST Commands

REST commands – also called HTTP methods – define the action we want to perform on the resource specified via the URI.

REST command	Description	Abbreviation	SQL
POST	Create or add new entries	C	INSERT
GET	Read, retrieve, search or view existing entries	R	SELECT
PUT or PATCH	Update or edit existing entires	U	UPDATE
DELETE	Delete or deactivate existing entries	D	DELETE

Table 2: Most important REST commands and their corresponding meanings

3.2 URI

A Uniform Resource Identifier (URI) is a string of characters, which is used to identify a specific resource. In a programmer's everyday life, said URI would mostly correspond to a URL, since the

access mechanism (scheme) – usually HTTP or HTTPS – is specified, too. This implies, that the set of URLs is a real subset of the set of URIs, as shown in Figure 2. Therefore, every URL is a URI, but not every URI is a URL.

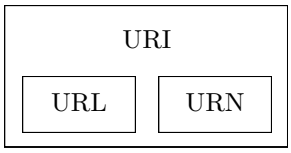

Figure 2: Relation between URI, URL and URN

URLs consists of four different parts. They usually start with the scheme or protocol definition. Since we want to talk to a service on the internet, we are going to use HTTP or HTTPS most of the time. The protocol definition is then followed by the hostname and the port number. In many cases, there is no need to explicitly specify the port number, since most APIs are bound to the corresponding ports for HTTP (80) or HTTPS (443). The final part of a URL is the resource path, which uniquely identifies a resource within the hostname.

Figure 3 shows an example URL, which identifies the sessions endpoint for our iOS Intro API.

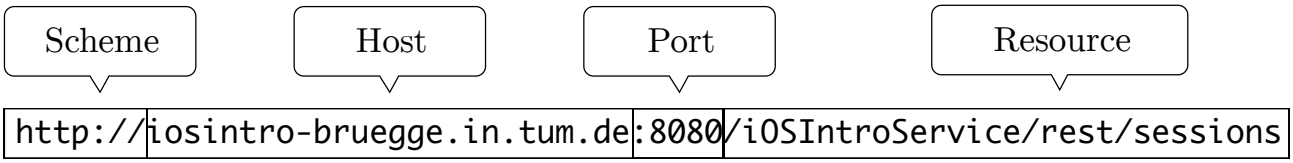

Figure 3: Sessions endpoint URL specification

By combining the REST command and the URI, we get the resulting REST command.

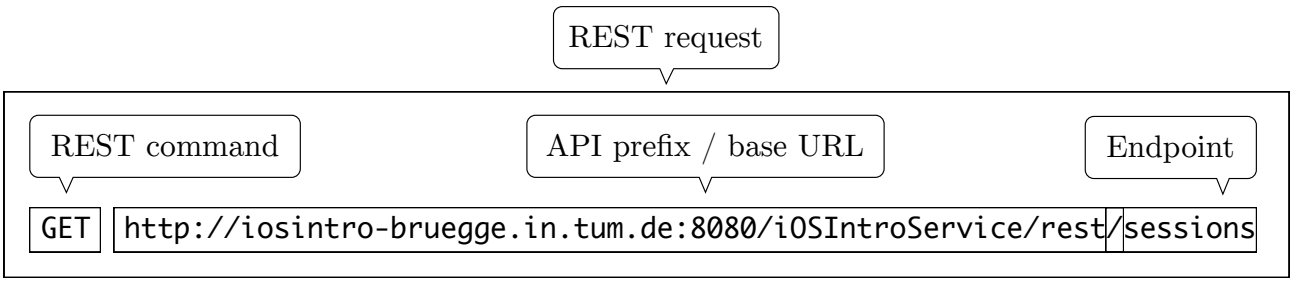

Figure 4: Anatomy of a REST request

4 RESTful endpoint design

4.1 Basics

Everyone who is building and interacting with RESTful services should be familiar with the basics of endpoint design. When deciding on how to model an endpoint it is crucial to understand how the objects we want to model relate to each other. Knowing the database schema and the relations between the tables and/or classes can be very helpful at this stage.

- Data operation on a single element: If you want to expose an endpoint to an object that only exists once, then you should model it as follows:
  - Pattern: `/element`
  - Example: /status
    Some API's expose a status endpoint, which can return information about the current status of the API (rate limiting statistics, load, uptime, etc.)

- Data operation on a collection: If you want to expose an endpoint to an entire collection of objects, then you should model it as follows:
  - Pattern: `/collection`
  - Example: /sessions
    The iOS-Intro API provides this endpoint. It represents a list of all sessions in this iOS-Intro course.

  Note: Always use the plural form to model collection endpoints. The plural is more appropriate, since collections always return a list of objects. Even if the endpoint only returns a single object, it is still a collection (which only contains a single object).

- Data operation on a single element of a collection: If you want to expose an endpoint to a single, specific object of a collection, then you should model it as follows:
  - Pattern: `/collection/:primaryIdentifier`
  - Example: /sessions/5
    The iOS-Intro API provides this endpoint. It represents a single session, which is identified by the ID 5.

- Chaining:
  - Pattern: `/coll1/:primC1Identifier/coll2/:primC2Identifier/coll-or-elem`
  - Example: /sessions/5/timeslots/27/registrations
    The iOS-Intro API provides this endpoint. It represents all registrations for timeslot 27 of session 5.

4.2 Best practices

When implementing API endpoints, it is important to consider some best practices. You should

- GET on a collection or an element in order to retrieve the collection or the element.
- POST to a collection in order to add the element. Do not PUT to a collection in order to add new elements.
- PUT or PATCH on an element to update the element. Do not POST to a element in order to update it.
- DELETE on a collection or an element in order to delete the collection or the element.

At this point, it is important to introduce the difference between PUT and PATCH. You should use PUT if you want the user to submit an entire object representation passed via the request body. On the other hand, you should use PATCH if you want the user to only submit the updates fields passed in the request body. PUTting to a user object, for example, would require to pass an entire user object, even if the user has only updated their password. By using PATCH you could just pass along the updated password. Most APIs only support PUT, since implementing PATCH on the server side is considered expensive because every single case needs to be covered.

4.3 Responses and body data

Now, we still need to take a look at the responses we get back from the server and the body data we have to submit to the server. When performing a

- GET request, the collection or the element specified via the URI gets returned and we do not need to specify any body data in the request.
- POST request, the newly inserted element gets returned. Usually, the server is responsible for generating a unique identifier for each object. Since we want to assign this ID to the object on the client side as well, the server returns this exact ID along with the created object. Of course, we need to submit the element we want to create via the request body.
- PUT request, the updated element gets returned. As with POST requests, we have to pass along the updated object via the request body.
- DELETE request, back an empty body gets returned. Likewise, we do not need to specify any request body data.

4.4 Example endpoints

In order to get you more familiar with API endpoint design, you should take a look at the following table, which is showing all endpoints of the iOS-Intro API. **Note**: Some endpoints are protected by an authentication mechanism.

```
Prefix: http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/
```

Action	Endpoint	Description
GET	/sessions	Retrieve all sessions
POST	/sessions	Add a new session
DELETE	/sessions	Delete all sessions
GET	/sessions/:sID	Retrieve session with ID :sID
PUT	/sessions/:sID	Update session with ID :sID
DELETE	/sessions/:sID	Delete session with ID :sID
GET	/sessions/:sID/timeslots	Retrieve all timeslots for session with ID :sID
POST	/sessions/:sID/timeslots	Add new timeslots for session with ID :sID
DELETE	/sessions/:sID/timeslots	Delete all timeslots for session with ID :sID

GET	/sessions/:sID/timeslots/:tID	Retrieve timeslot wit ID :tID for session with ID :sID
PUT	/sessions/:sID/timeslots/:tID	Update timeslot wit ID :tID for session with ID :sID
DELETE	/sessions/:sID/timeslots/:tID	Delete timeslot wit ID :tID for session with ID :sID
GET	/sessions/:sID/timeslots/:tID/registrations	Retrieve all registrations for timeslot wit ID :tID for session with ID :sID
POST	/sessions/:sID/timeslots/:tID/registrations	Add new registration for timeslot wit ID :tID for session with ID :sID
DELETE	/sessions/:sID/timeslots/:tID/registrations	Delete all registrations wit ID :tID for session with ID :sID
GET	/students	Retrieve all students
POST	/students	Add a new student
DELETE	/students	Delete all students
GET	/students/:sID	Retrieve students with ID :sID
PUT	/students/:sID	Update student with ID :sID
DELETE	/students/:sID	Delete students with ID :sID
GET	/rooms	Retrieve all rooms
POST	/rooms	Add a new room
DELETE	/rooms	Delete all rooms
GET	/rooms/:rID	Retrieve rooms with ID :rID
PUT	/rooms/:rID	Update room with ID :rID
DELETE	/rooms/:rID	Delete rooms with ID :rID

4.5 Exercise 1

Specify a REST request, which queries an endpoint for a list of all students that registered for the iOS Praktikum (and therefore for this Intro Course). Use the syntax introduced in the previous chapter.

```
Syntax: REST_COMMAND PREFIX/ENDPOINT
```

```
Prefix: http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/
```

4.6 Solution 1

```
GET http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/students
```

4.7 Query parameters

Query parameters are often used to add supplemental information to the request, without providing data inside of the request body. The syntax is quite straight forward: You can simply add additional parameters as key-value pairs to the request line. The first key-value pair is indicated by a leading `?`, each other key-value pair is indicated by a leading `&`. Key and value are separated by an `=`.

```
http://example.com/endpoint?field1=value1&field2=value2&field3=value3...
```

4.8 Exercise 2

Specify a REST request, which updates the field `email` of the student with ID `0437` to `js@in.tum.de`. Note that calls to that endpoint now require authentication in order to prevent abuse. The auth-code we got from the system is `5sZym`. Add it as the value for the `oauth_token` parameter.

The JSON representation of the user object, looks like this:

```
{
  "email":"smithj@in.tum.de",
  "identifier":"0437",
  "name":"John Smith"
}
```

```
Syntax: REST_COMMAND PREFIX/ENDPOINT
```

```
Prefix: http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/
```

4.9 Solution 2

```
PUT http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/students/0437?
    oauth_token=5sZym
```

```
{
  "email":"js@in.tum.de",
  "identifier":"0437",
  "name":"John Smith"
}
```

5 REST responses

When the server has finished gathering all of the requested data, it can compose a HTTP response and send it back to the client. Table 4 shows the key components of such a response.

HTTP Response	Status Line
	Headers
	Message Body Data

Table 4: HTTP Response

5.1 HTTP status codes

Whenever the server builds up a response for the client, a HTTP status code is specified in the response header. HTTP status codes, therefore, give you some initial information about the response. Based on the HTTP status code, we can, for example, decide whether the request was successful or not and who we can blame in case of failure.

HTTP codes are grouped in five different classes, which are specified by the first digit of the status code.

5.1.1 Informational - 1xx

This class of status code indicates a provisional response, consisting only of the status line and optional headers.

Class	HTTP code	Message
Informational - 1xx	100	Continue
	101	Switching Protocols
	103	Checkpoint

Table 5: Informational HTTP status codes

5.1.2 Successful - 2xx

This class of status code indicates that the client's request was successfully received, understood, and accepted.

Class	HTTP code	Message
Successful - 2xx	200	OK
	201	Continue
	202	Accepted
	203	Non-Authoritative Information
	204	No Content
	205	Reset Content
	206	Partial Content

Table 6: Successful HTTP status codes

5.1.3 Redirection - 3xx

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

Class	HTTP code	Message
Redirection - 3xx	300	Multiple Choices
	301	Moved Permanently
	302	Found
	303	See Other
	304	Not Modified
	306	Switch Proxy
	307	Temporary Redirect
	308	Resume Incomplete

Table 7: Redirection HTTP status codes

5.1.4 Client Error - 4xx

This class of status code indicates that the server could not fulfill the request, because the client's request contained some errors or wrong information.

Class	HTTP code	Message
Client Error - 4xx	400	Bad Request
	401	Unauthorized
	402	Payment Required
	403	Forbidden
	404	Not Found
	405	Method Not Allowed
	406	Not Acceptable
	407	Proxy Authentication Required
	408	Request Timeout
	409	Conflict
	410	Gone
	411	Length Required
	412	Precondition Failed
	413	Request Entity Too Large
	414	Request-URI Too Long
	415	Unsupported Media Type
	416	Requested Range Not Satisfiable
	417	Expectation Failed

Table 8: Client error HTTP status codes

5.1.5 Server Error - 5xx

This class of status code indicates that the server encountered an unexpected condition, which prevented it from fulfilling the request.

Class	HTTP code	Message
Server Error - 5xx	500	Internal Server Error
	501	Not Implemented
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Timeout
	505	HTTP Version Not Supported
	511	Network Authentication Required

Table 9: Server error HTTP status codes

5.2 Response Body

The response body contains objects mapped to a specific data format, which would be JSON or XML in most cases. Both data formats basically consist of key-value pairs, which can store a variety of basic data types.

An example response for

```
GET http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/sessions
```

would look like this:

```
[
  {
```

```
    "descriptionText":"Introduction to the iOS Praktikum.",
    "identifier":"1",
    "name":"Introduction",
    "timeslots": [
      "4",
      "5",
      "6"
    ]
  },
  {
    "descriptionText":"Introduction to the Swift language.",
    "identifier":"2",
    "name":"Introduction to Swift",
    "timeslots":[]
  }
]
```

5.3 Exercise 3

Execute the following REST requests in your REST console or browser of choice. Which status code do you get? Apigee[6] provides a free web-based console, which fits our needs.

```
GET http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/session
```

```
GET http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/sessions
```

5.4 Solution 3

The first request should return a `404` error, since this endpoint is not implemented on the server. The second request should return a JSON-formatted list containing all sessions.

6 REST requests with AFNetworking

AFNetworking is an open source, easy to use, high level networking library developed by Mattt Thompson. It is build on top of Apple's own `NSURLConnection` (until mid 2013) and `NSURLSession` (after mid 2013). Thanks to AFNetworking's built in serializers for JSON, XML and Apple's own plist[7], objects can be converted to and from these formats quite easily.

6.1 Basic setup

In order to be able to use AFNetworking in our application, we need to perform some basic setup. Since we do not want to perform this setup before every request, this code should be put inside the initialization method.

1. We need to setup the base URL or prefix for our API as follows:

```
1  let apiPrefix = "http://iosintro-bruegge.in.tum.de:8080/iOSIntroService/rest/"
2  let apiPrefixURL = NSURL(string: apiPrefix)
```

[6] https://apigee.com/console/others
[7] https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man5/plist.5.html

2. We create a AFHTTPRequestOperationManager object and initialize it with the corresponding prefix.

```
let manager = AFHTTPRequestOperationManager(baseURL: apiPrefixURL)
```

3. We want to set the request and response serializers to JSON, which can be done as follows:

```
manager.requestSerializer = AFJSONRequestSerializer()
manager.responseSerializer = AFJSONResponseSerializer()
```

6.2 GET

6.2.1 Example

After the initial setup, we are now ready to perform our first GET request via AFNetworking. In the first example we are going to retrieve a list of all students, that are currently registered for the iOS-Intro course.

We simply call the `GET` function on the `manager` and provide the `"students"` endpoint as the first parameter. The `success` closure provides us with the `responseObject`, which we need to cast to an appropriate object representation (in our case an array of dictionaries). The `Student` class offers a class method to convert the array of dictionaries to an array of `Student` object. At the end, we pass the array of students to the outer `success` closure. It is also good practice to always handle errors gracefully, so we pass it to the outer `failure` closure, too.

```
func retrieveStudents(success: (students:[Student]) -> Void, failure: (error : NSError
    !) -> Void) {
  manager.GET("students", parameters: nil, success: { (operation, responseObject) in
      let studentArray = responseObject as [[String : AnyObject]]
      println(studentArray)
      let students = Student.studentsFromArray(studentArray)
      success(students: students)
  }, failure: { (operation, error) in
      failure(error: error)
  })
}
```

6.2.2 Exercise 4

Retrieve all sessions from this intro course by using AFNetworking, print the response array and call the method's success closure with the generated sessions.

To do that, please follow these steps:

1. Open the example app's Xcode project.
2. Navigate to the `DataManager.swift` file.
3. Complete the function `retrieveSessions(success: (courses:[CourseType]) -> Void, failure: (error : NSError!) -> Void)`.
4. You can use the `coursesFromArray` convenience method I've implemented as a class function on `CourseType`.
5. Build and run the app in the simulator and tap the "Fetch sessions" button.

6.2.3 Solution 4

```
func retrieveSessions(success: (courses:[CourseType]) -> Void, failure: (error :
    NSError!) -> Void) {
  manager.GET("sessions", parameters: nil, success: { (operation, responseObject) in
      let sessionArray = responseObject as [[String : AnyObject]]
      println(sessionArray)
      let sessions = CourseType.coursesFromArray(sessionArray)
      success(courses: sessions)
  }, failure: { (operation, error) in
      failure(error: error)
  })
}
```

6.3 POST

6.3.1 Example

In the second example we are going to add a new student.

A `POST` request is a bit more complicated compared to a `GET` request, since we have to add the object we want to create in the request body. Additionally, we also have to provide authentication credentials this time. These credentials are mandatory for all requests which are able to perform data manipulation on the server (`POST`, `PUT`, `DELETE`).

First, we need to construct the endpoint by providing the appropriate user credentials. Then, we create the student, which we want to submit to the server. To do that, we still have to convert the student object to a dictionary, which is recognized as valid body data by AFNetworking. To execute the request, we call the `POST` function on the `manager` and pass along the endpoint and the student dictionary. As a response, the server returns the newly created student. Therefore, we have to adapt the response parsing and casting slightly. This time, the server returns an array of dictionaries, but simply a dictionary, which represents the created student.

```
func addStudent(success: (student:Student) -> Void, failure: (error : NSError!) ->
    Void) {
    let endpoint = "students?username=\(self.username)&password=\(self.pw)"
    let student = Student(username: "steve", password: "supersecret")
    manager.POST(endpoint, parameters: student.dictFromStudent(), success: { (
        operation, responseObject) in
        let studentDict = responseObject as [String : AnyObject]
        println(studentDict)
        let student = Student.studentFromDictionary(studentDict)
        success(student: student)
    }, failure: { (operation, error) in
        failure(error: error)
    })
}
```

6.3.2 Exercise 5

Create a new session with a random title and a random `descriptionText` for this intro course by using AFNetworking, print the response dictionary and call the method's success closure with the generated session. **Note**: Auth is required: `username = "johnsnow"` and `password = "v4f6L6L89y"`. Memorize the value of the identifier field you get back in the response.

To do that, please follow these steps:

1. Complete the function `addSession(success: (course:CourseType) -> Void, failure: (error : NSError!) -> Void)`.

2. Do not forget to add the authentication query parameters to your API call.

3. You can use the `dictFromCourseType` and `courseFromDictionary` convenience methods I've implemented on `CourseType`.

4. Build and run the app in the simulator and tap the "Add new session" button.

6.3.3 Solution 5

```
let endpoint = "sessions?username=\(self.username)&password=\(self.pw)"
let session = CourseType(name: "Core Animation", description: "Wow. Such smooth.")
manager.POST(endpoint, parameters: session.dictFromCourseType(), success: { (operation
    , responseObject) in
    let sessionDict = responseObject as [String : AnyObject]
    println(sessionDict)
    let session = CourseType.courseFromDictionary(sessionDict)
    success(course: session)
}, failure: { (operation, error) in
    failure(error: error)
})
```

6.4 PUT

6.4.1 Example

In the third example we are going to update the password of the student we have created earlier.

First, we need to construct the endpoint by providing the student identifier and the appropriate user credentials. Then, we create a new student, which we want to submit to the server. This time, we also have to provide the student's identifier in the initializer, since the server needs to know which student we are going to update. To execute the request, we call the `PUT` function on the `manager` and pass along the endpoint and the student dictionary. As a response, the server returns the updated student. The parsing and casting of the response is basically the same as with `POST`.

```
func updateStudent(success: (student:Student) -> Void, failure: (error : NSError!) ->
    Void) {
    let identifier = 42
    let endpoint = "students/\(identifier)?username=\(self.username)&password=\(self.
        pw)"
    let student = Student(identifier: "\(identifier)", username: "steve", password: "
        topsecret")
    manager.PUT(endpoint, parameters: student.dictFromStudent(), success: { (operation
        , responseObject) in
        let studentDict = responseObject as [String : AnyObject]
        println(studentDict)
        let student = Student.studentFromDictionary(studentDict)
        success(student: student)
    }, failure: { (operation, error) in
        failure(error: error)
    })
}
```

6.4.2 Exercise 6

Update the session you've just created to a different `descriptionText` by using AFNetworking, print the response dictionary and call the method's success closure with the generated session. **Note**: Auth is required: `username = "johnsnow"` and `password = "v4f6L6L89y"`.

To do that, please follow these steps:

1. Complete the function `updateSession(success: (course:CourseType) -> Void, failure: (error : NSError!) -> Void)`
2. Do not forget to add the authentication query parameters to your API call
3. You can use the `dictFromCourseType` and `courseFromDictionary` convenience methods I've implemented on `CourseType`.
4. Build and run the app in the simulator and tap the "Update session" button

6.4.3 Solution 6

```
let identifier = 54
let endpoint = "sessions/\(identifier)?username=\(self.username)&password=\(self.pw)"
let session = CourseType(identifier: "\(identifier)", name: "Core Animation",
    description: "Fancy animations.")
manager.PUT(endpoint, parameters: session.dictFromCourseType(), success: { (operation,
     responseObject) in
    let sessionDict = responseObject as [String : AnyObject]
    println(sessionDict)
    let session = CourseType.courseFromDictionary(sessionDict)
    success(course: session)
}, failure: { (operation, error) in
    failure(error: error)
})
```

6.5 DELETE

6.5.1 Example

In the last example we are going to delete the student we have created earlier.

Again, we have to construct the endpoint by providing the student identifier and the authentication credentials. Then, we call the `DELETE` function on the `manager`. Since `DELETE` calls do not return any body data in the response, there is no need to parse the response and we can just call the `success` closure.

```
func deleteStudent(success: () -> Void, failure: (error : NSError!) -> Void) {
    let identifier = 42
    let endpoint = "students/\(identifier)?username=\(self.username)&password=\(self.
        pw)"
    manager.DELETE(endpoint, parameters: nil, success: { (operation, responseObject)
        in
        success()
    }, failure: { (operation, error) in
        failure(error: error)
    })
}
```

6.5.2 Exercise 7

Delete the session you've created earlier by using AFNetworking and call the method's success closure.
Note: Auth is required: `username = "johnsnow"` and `password = "v4f6L6L89y"`.

To do that, please follow these steps:

1. Complete the function `deleteSession(success: () -> Void, failure: (error : NSError!) -> Void)`
2. Do not forget to add the authentication query parameters to your API call
3. Build and run the app in the simulator and tap the "Delete session" button

6.5.3 Solution 7

```
let identifier = 54
let endpoint = "sessions/\(identifier)?username=\(self.username)&password=\(self.pw)"
manager.DELETE(endpoint, parameters: nil, success: { (operation, responseObject) in
    success()
}, failure: { (operation, error) in
    failure(error: error)
})
```

7 Backend processing

Whenever the server receives a request from a client some kind of backend processing takes place. For this iOS Introduction Course's `RegistrationApp` we are using a server developed in Java (J2EE), which runs on a Glassfish[8] 4.0 application server. The Jersey REST interface handles endpoint mapping for us. Endpoints are automatically mapped to dedicated method calls in the resource classes. At the storage level we are using MySQL as a relational database and JPA is used for persistence.

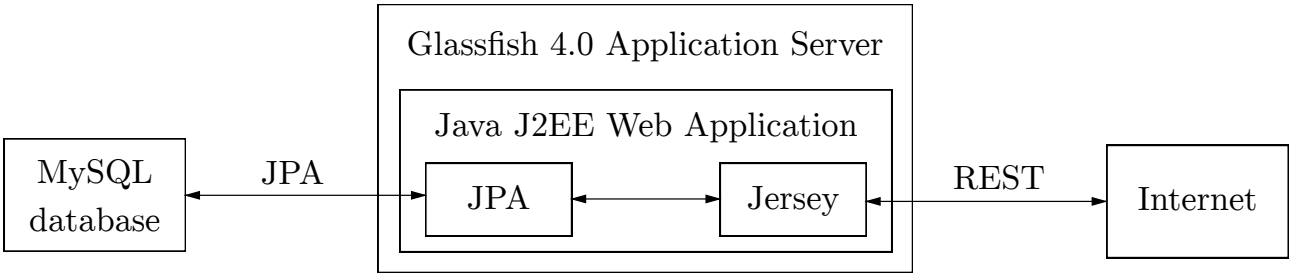

Figure 5: General server architecture

7.1 Resources and endpoints

Thanks to Jersey, we can easily connect endpoints to corresponding resource classes.

In this particular example, the `@Path("/sessions")` annotation indicates, that the class `SessionsResource` is connected to the `/sessions` endpoint. In order to be able to communicate to the database, we have access to a `PersistenceManager` instance, which is managed by JPA.

```
@Path("/sessions")
public class SessionsResource {

  @EJB private PersistenceManager persistenceManager;

  public SessionsResource()
```

[8] `https://glassfish.java.net`

```
7   {
8   }
9
10  // ...
```

Actual REST request are then passed to a method corresponding to the request's properties. A `GET` on sessions would, therefore, lead to a call of the `getSessions()` method, since the latter is annotated with a `@GET` and there is no additional `@Path` annotation. To indicate the produced media type, we still have to set the `@Produces` annotation to our desired value: `MediaType.APPLICATION_JSON`. In `POST` and `PUT` methods, we need to specify the expected media type for the request body, which can be done by setting the `@Consumes` annotation. Query parameters can be extracted by adding the `@PathParam` annotation to parameters provided in the corresponding method's signature. In our case, we need a username-password combination passed via query parameters, in order to give the user access to `POST`, `PUT` and `DELETE` endpoints. Additional endpoints which want to extend from `/sessions` need to specify an additional `@Path`. The provided string can contain parameters, which have to be surrounded by curly braces.

```
1   // ...
2
3   //=========================================================
4   // Sessions
5   //=========================================================
6
7   @GET
8   @Produces(MediaType.APPLICATION_JSON)
9   public Collection<CourseSession> getSessions() {
10    System.out.println("GET on /sessions");
11    return getAllSessions();
12  }
13
14  @POST
15  @Produces(MediaType.APPLICATION_JSON)
16  @Consumes(MediaType.APPLICATION_JSON)
17  public CourseSession createSession(CourseSession session, @QueryParam("username")
        String name, @QueryParam("password") String password) {
18    boolean loggedIn = persistenceManager.checkLogin(name, password);
19    if (loggedIn) {
20      persistenceManager.persistObject(session);
21      return session;
22    } else {
23      throw new NotAuthorizedException("Unauthorized");
24    }
25  }
26
27  @GET
28  @Path("{sessionId}")
29  @Produces(MediaType.APPLICATION_JSON)
30  public CourseSession getSession(@PathParam("sessionId") String sessionId,
        @QueryParam("username") String name, @QueryParam("password") String password) {
31    return findSession(sessionId);
32  }
33
34  @PUT
35  @Path("{sessionId}")
36  @Produces(MediaType.APPLICATION_JSON)
37  @Consumes(MediaType.APPLICATION_JSON)
38  public CourseSession updateSession(@PathParam("sessionId") String sessionId,
        CourseSession session, @QueryParam("username") String name, @QueryParam("
        password") String password) {
```

```
39      System.out.println("PUT␣on␣/sessions/" + sessionId);
40      System.out.println(session.getIdentifier() + "␣|␣" + session.getName() + "␣|␣" +
            session.getDescriptionText());
41      boolean loggedIn = persistenceManager.checkLogin(name, password);
42      if (loggedIn) {
43        persistenceManager.updateObject(session);
44        return session;
45      } else {
46        throw new NotAuthorizedException("Unauthorized");
47      }
48    }
49
50    @DELETE
51    @Path("{sessionId}")
52    @Produces(MediaType.APPLICATION_JSON)
53    public CourseSession deleteSession(@PathParam("sessionId") String sessionId,
          CourseSession session, @QueryParam("username") String name, @QueryParam("
          password") String password) {
54      boolean loggedIn = persistenceManager.checkLogin(name, password);
55      if (loggedIn) {
56        persistenceManager.deleteObject(sessionId, CourseSession.class);
57        return session;
58      } else {
59        throw new NotAuthorizedException("Unauthorized");
60      }
61    }
62
63    //================================================
64    // Timeslots
65    //================================================
66
67    @GET
68    @Path("{sessionId}/timeslots")
69    @Produces(MediaType.APPLICATION_JSON)
70    public Collection<Timeslot> getTimeslotsForSession(@PathParam("sessionId") String
          sessionId, @QueryParam("username") String name, @QueryParam("password") String
          password) {
71      return findSession(sessionId).getTimeslots();
72    }
73
74    @POST
75    @Path("{sessionId}/timeslots")
76    @Produces(MediaType.APPLICATION_JSON)
77    @Consumes(MediaType.APPLICATION_JSON)
78    public Timeslot createTimeslotForSession(@PathParam("sessionId") String sessionId,
          Timeslot timeslot, @QueryParam("username") String name, @QueryParam("password")
          String password) {
79      boolean loggedIn = persistenceManager.checkLogin(name, password);
80      if (loggedIn) {
81        CourseSession session = findSession(sessionId);
82        if (session == null) {
83          throw new NotFoundException("Session␣not␣found");
84        }
85        session.getTimeslots().add(timeslot);
86        persistenceManager.persistObject(timeslot);
87        return timeslot;
88      } else {
89        throw new NotAuthorizedException("Unauthorized");
90      }
91    }
92
```

```
93  @GET
94  @Path("{sessionId}/timeslots/{timeslotId}")
95  @Produces(MediaType.APPLICATION_JSON)
96  public Timeslot getTimeslotForSession(@PathParam("sessionId") String sessionId,
        @PathParam("timeslotId") String timeslotId, @QueryParam("username") String name,
         @QueryParam("password") String password) {
97    return findTimeslot(timeslotId);
98  }
99
100 @PUT
101 @Path("{sessionId}/timeslots/{timeslotId}")
102 @Produces(MediaType.APPLICATION_JSON)
103 @Consumes(MediaType.APPLICATION_JSON)
104 public Timeslot updateTimeslot(@PathParam("sessionId") String sessionId, @PathParam(
        "timeslotId") String timeslotId, Timeslot timeslot, @QueryParam("username")
        String name, @QueryParam("password") String password) {
105   boolean loggedIn = persistenceManager.checkLogin(name, password);
106   if (loggedIn) {
107     persistenceManager.updateObject(timeslot);
108     return timeslot;
109   } else {
110     throw new NotAuthorizedException("Unauthorized");
111   }
112 }
113
114 //=========================================================
115 // Registrations
116 //=========================================================
117
118 @GET
119 @Path("{sessionId}/timeslots/{timeslotId}/registrations")
120 @Produces(MediaType.APPLICATION_JSON)
121 public Collection<Registration> getRegistrationsForTimeslotForSession(@PathParam("
        sessionId") String sessionId, @PathParam("timeslotId") String timeslotId,
        @QueryParam("username") String name, @QueryParam("password") String password) {
122   boolean loggedIn = persistenceManager.checkLogin(name, password);
123   if (loggedIn) {
124     Timeslot timeslot = findTimeslot(timeslotId);
125     if (timeslot == null) {
126       throw new NotFoundException("Timeslot not found");
127     }
128     return timeslot.getRegistrations();
129   } else {
130     throw new NotAuthorizedException("Unauthorized");
131   }
132 }
133
134 @POST
135 @Path("{sessionId}/timeslots/{timeslotId}/registrations")
136 @Produces(MediaType.APPLICATION_JSON)
137 @Consumes(MediaType.APPLICATION_JSON)
138 public Registration createRegistrationForTimeslotForSession(@PathParam("sessionId")
        String sessionId, @PathParam("timeslotId") String timeslotId, Registration
        registration, @QueryParam("username") String name, @QueryParam("password")
        String password) {
139   boolean loggedIn = persistenceManager.checkLogin(name, password);
140   System.out.println("Logged in: " + loggedIn);
141   System.out.println("name: " + name + "\nname in registration: " + registration.
        getStudent().getName());
142   if (loggedIn && name.equals(registration.getStudent().getName())) {
143     Timeslot timeslot = findTimeslot(timeslotId);
```

```
144         if (timeslot == null) {
145           throw new NotFoundException("Timeslot not found");
146         }
147
148         timeslot.getRegistrations().add(registration);
149
150         Student student = findStudent(registration.getStudent().getIdentifier());
151         if (student == null) {
152           throw new NotFoundException("Student not found");
153         }
154         student.addRegistrations(registration);
155
156         persistenceManager.persistObject(registration);
157         persistenceManager.updateObject(timeslot);
158         persistenceManager.updateObject(student);
159
160         return registration;
161       } else {
162         throw new NotAuthorizedException("Unauthorized");
163       }
164     }
165
166     // ...
```

At the end of this class we still have some private helper methods, which are querying the persistence context in different ways.

```
1     // ...
2
3     //=========================================================
4     // Private helpers
5     //=========================================================
6
7     private List<CourseSession> getAllSessions() {
8       return persistenceManager.findAll(CourseSession.class);
9     }
10
11    private CourseSession findSession(String sessionId) {
12      return persistenceManager.find(CourseSession.class, sessionId);
13    }
14
15    private Timeslot findTimeslot(String timeslotId) {
16      return persistenceManager.find(Timeslot.class, timeslotId);
17    }
18
19    private Student findStudent(String studentId) {
20      return persistenceManager.find(Student.class, studentId);
21    }
22
23  }
```

7.2 Models

Just like in our iOS application, models are a key component on the server, too.

The `@Entity` and `@XmlRootElement` annotations at the top of the model declaration indicate that this model should be managed as a persistence entity. `@Column` annotations are used to indicate database columns. The primary identifier is determined by the `@Id` annotation and since we do not want to manage the primary ID ourselves we add the additional `@GeneratedValue` annotation.

Relations between tables can be mapped by using `@OneToOne`, `@OneToMany` or `@ManyToMany` annotations. Keep in mind that such relations need to be defined in both model classes since they are not bidirectional by default.

```
@Entity
@XmlRootElement
public class CourseSession implements Serializable {

  private static final long serialVersionUID = -4721548810711003689L;

  @Id
  @GeneratedValue
  private String identifier;

  @Column
  private String name;

  @Column
  private String descriptionText;

  @OneToMany(mappedBy="session", fetch=FetchType.EAGER)
  private List<Timeslot> timeslots;

  public CourseSession() {
  }

  public CourseSession(String title, String descriptionText) {
    this.name = title;
    this.descriptionText = descriptionText;
    this.timeslots = new ArrayList<Timeslot>();
  }

  // Getters and setters

  // ...

}
```

8 Final exercise

Since sessions take place at a specific location, we have to manage rooms as well. The API provides two dedicated endpoints (`/rooms` and `/rooms/:rID`), which can be used to retrieve, add, update and delete rooms. Again, you should complete all function inside of the `DataManager.swift` file. **Note:** Calling the success closures is optional this time, since we did not provide you with dedicated data serialization methods. Interested students can take this as an optional challenge.

`Good luck and happy coding! :)`