# API

## Table of Contents



# EDAF20 Project - Web Server API

You will in this part implement a REST API for a subset of the Krusty database. You are given a skeleton project that you can start from (download **krusty-skeleton.zip (https://canvas.education.lu.se/courses/10339/files/1051624/download?wrap=1)** ).

# Background

A REST server (or REST service) is a program which runs on a computer, waits for requests, and sends back responses – normally it uses the HTTP(s) protocol for its communication. In lecture 7, REST was introduced and we looked at a REST server that handled information about bank accounts (see **slides (https://canvas.education.lu.se/courses/10339/files/1046887/download?wrap=1)** and **example project (https://canvas.education.lu.se/courses/10339/files/1046891/download?wrap=1)** for bank accounts).

The REST server will be implemented using the framework **Spark (http://sparkjava.com/)** (shown during a lecture) and the responses will be encoded as **JSON objects (https://en.wikipedia.org/wiki/JSON)** .

# Installation

Download the provided Gradle project (`krusty-skeleton.zip`) and unzip it, and then import it into Eclipse (or your favorite IDE). A project can be imported in Eclipse by `File -> Import... -> Gradle -> Existing Gradle Project... -> Next` (or similar), and then select the root directory as the unzipped directory.

The project is a Gradle project, which makes dependencies to external libraries easy to manage. This will make Eclipse to download all dependencies when opening the project automatically. This is also why the directory structure looks like it does.

# Assignment

You will implement the following REST endpoints:

```
  GET /api/v1/customers   // Get all customers
  GET /api/v1/raw-materials // Get all raw materials
  GET /api/v1/cookies     // Get all cookies
  GET /api/v1/recipes     // Get all recipes
  GET /api/v1/pallets     // Get all pallets - uses query parameters for filtering

  POST /api/v1/reset      // Reset the database - see below for details
  POST /api/v1/pallets    // Create a pallet - uses query parameter for specifying co
okie
```

The given project contains the following classes:

- **`main.java.krusty.ServerMain`**
  Defines the REST endpoints using the framework Spark. The endpoints are delegated to the class `Database`.
- **`main.java.krusty.Database`**
  Defines methods for accessing the database and returning JSON objects (as Strings). The skeleton project contains empty methods that you should implement.

- `main.java.krusty.JSONizer`
  An auxiliary class that takes a `ResultSet` and automatically converts it to a JSON object (as a String).
- `test.java.krusty.KrustyTests`
  Automatic test cases for the REST endpoints. You can use the tests to verify that your implementation is correct.

The endpoints are already defined in the skeleton project and in the class `ServerMain`:

```
  public static final String API_ENTRYPOINT = "/api/v1";

  private void initRoutes() {
     get(API_ENTRYPOINT + "/customers", (req, res) -> db.getCustomers(req, res));
     get(API_ENTRYPOINT + "/raw-materials", (req, res) -> db.getRawMaterials(req, re
s));
     get(API_ENTRYPOINT + "/cookies", (req, res) -> db.getCookies(req, res));
     get(API_ENTRYPOINT + "/recipes", (req, res) -> db.getRecipes(req, res));
     get(API_ENTRYPOINT + "/pallets", (req, res) -> db.getPallets(req, res));

     post(API_ENTRYPOINT + "/reset", (req, res) -> db.reset(req, res));
     post(API_ENTRYPOINT + "/pallets", (req, res) -> db.createPallet(req, res));
  }
```

As can be seen, the endpoints are delegated to corresponding methods in the `Database` class. **Your task is to implement these methods.** For each endpoint, a request (`req`) and a response (`res`) object are passed to the database method. You will use the request object when handling query parameters, which are used for the endpoints `GET /api/v1/pallets` and `POST /api/v1/pallets`. The response object is not needed for any endpoint, but it can be used for setting different HTTP status codes, for example.

A common advice when coding is to work iteratively, thus, working in small steps and testing the code after each step. The provided project includes automatic tests that you can run to test your implementation, see **Automatic Testing (https://fileadmin.cs.lth.se/cs/Education/EDAF20/2020/project/api.html#automatic-testing)**. You can also test your implementation manually, which might be more useful when you start your implementation, see **Manual Testing (https://fileadmin.cs.lth.se/cs/Education/EDAF20/2020/project/api.html#manual-testing)**.

There is already a web app that uses the endpoints to provide a user interface, see **Web App (https://fileadmin.cs.lth.se/cs/Education/EDAF20/2020/project/api.html#web-app)**. You can try this out when you have implemented the endspoints. The class `ServerMain` enables something called **CORS (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS)** (Cross-Origin Resource Sharing), which allows the web app to access the endpoints. However, you do not need to worry about this.

# REST Endpoints

This section describes each endpoint in more detail, and gives implementation advices for some of the endpoints.

## Endpoint `GET /api/v1/customers`

Returns all customers, where each customer is described with a `name` and an `address`. Example response:

```
{   "customers": [
      {"name": "Bjudkakor AB", "address": "Ystad"},
     ...
      ]
}
```

Note that JSON properties are case-sensitive, meaning that you should use `"customers"` and not `"Customers"`, `"name"` and not `"Name"`, etc. It is important that the JSON object you create use the expected property names correctly, otherwise the automatic tests and the web app will fail.

As described before, you implement this endpoint by implementing the method `Database.getCustomers`. This method returns a JSON object represented as a String. The parameters `req` and `res` are not needed for this endpoint. When creating JSON responses, you can use the provided class `JSONizer`, which takes a `ResultSet` and returns a JSON object as a `String`. Example:

```
public String getCustomers(Request req, Response res) {
  ...
   try (...} {
    ...
    ResultSet resultSet = ...;
    String json = JSONizer.toJSON(resultSet, "customers");
    return json;
  } catch (...) {
    ...
  }
  ...
}
```

Consider the following SQL query:

```
> SELECT name, address FROM customers;
+----------------+--------------+
| name           | address      |
+----------------+--------------+
| Finkakor AB    | Helsingborg  |
| Småbröd AB     | Malmö        |
+----------------+--------------+
```

Assume that the result above is represented by a `ResultSet` and that we call `JSONizer.toJSON(resultSet, "customers")`. Then we get the following `String` back:

```
{
  "customers": [
    {"name": "Finkakor AB", "address": "Helsingborg"},
    {"name": "Småbröd AB", "address": "Malmö"}
 ]
}
```

The method `JSONizer.toJSON` uses the attribute names in the SQL query when translating the `ResultSet` to a JSON object. If your attribute names do not reflect the expected JSON response, then you can use SQL aliases. For

example, let's assume that your attributes are called `company_name` and `company_address`, then to use the `JSONizer` class, we can use aliases for them to create the expected JSON response:

```
SELECT
  company_name AS name,
  company_address as address
FROM customers
```

It is also possible to create the JSON object as a String without using the method `JSONizer.toJSON`, but this requires more effort.

## Endpoint `GET /api/v1/raw-materials`

Returns all raw materials, where each material has a `name`, `amount` and an `unit`. Example response:

```
{
  "raw-materials": [
    {"name": "Bread crumbs", "amount": 500000, "unit": "g"},
    {"name": "Egg whites", "amount": 500000, "unit": "ml"},
    ...
  ]
}
```

## Endpoint `GET /api/v1/cookies`

Returns all cookies, where each cookie is described by a `name`. Example response:

```
{
  "cookies": [
    {"name": "Amneris"},
    ...
  ]
}
```

## Endpoint `GET /api/v1/recipes`

Returns the recipes for all cookies. A recipe for a cookie is described by several entries (one entry for each ingredient). Example response:

```
{
  "recipes": [
    {"cookie": "Amneris", "raw_material": "Butter", "amount": 250, "unit": "g"},
    {"cookie": "Amneris", "raw_material": "Marzipan", "amount": 750, "unit": "g"},
    ...
    {"cookie": "Berliner", "raw_material": "Flour", "amount": 350, "unit": "g"},
    {"cookie": "Berliner", "raw_material": "Butter", "amount": 250, "unit": "g"},
    ...
  ]
}
```

# Endpoint `GET /api/v1/pallets`

Returns all produced pallets sorted by production date (newest first). Each pallet is described by an `id`, `cookie`, `production_date`, `customer` and `blocked`. Example response:

```
{
  "pallets": [
    {"id": 1, "cookie": "Amneris", "production_date": "2019-01-01 13:00:00", "custome
r":
       "Finkakor AB", "blocked": "no"},
    {"id": 2, "cookie": "Nut ring", "production_date": "2019-01-05 15:00:00",
      "customer": "null", "blocked": "no"},
    ...
  ]
}
```

It is also possible to filter pallets based on the query parameters `from`, `to`, `cookie`, `blocked`. Examples:

```
GET /pallets?from=2018-01-01  // Get all pallets produced after 2018-01-01 (inclusiv
e)
GET /pallets?to=2020-01-01    // Get all pallets produced before 2020-01-01 (inclusiv
e)
GET /pallets?cookie=Amneris   // Get all pallets with the cookie Amneris
GET /palelts?blocked=yes      // Get all blocked pallets
```

Note that these filters can be combined in arbitary ways, such as:

```
GET /pallets?from=2018-01-01&to=2020-01-01&cookie=Amneris
```

Query parameters can be obtained from the `Request` object that is passed as a parameter. For example,

```
public String getPallets(Request req, Response res) {
  ...
  if (req.queryParams("from") != null) {
    String from = req.queryParams("from");
    // Do something with "from" here
  }
  ...
}
```

This code checks if the query parameter `from` is specified by the user and then extracts the provided value.

Since the input is given by the user (as query parameters), we need to use prepared statements to protect against SQL injections. When prepared statements are used, the SQL query needs first to be created, and then the question marks are replaced with actual values using `setX` methods (like `setString`). Since search filters can be combined in arbitary ways, we need to dynamically build the SQL query and store the provided values somehow. One way to store the values is to use an `ArrayList` and then iterative over it to replace the question marks, which is illustrated in the following code.

```
public String getPallets(Request req, Response res) {
  String sql = ...;

  ArrayList<String> values = new ArrayList<String>();
  if (req.queryParams("from") != null) {
    sql += ...;
    values.add(req.queryParams("from"));
  }
  // Check other query parameters
  ...

  try (PreparedStatement stmt = connection.prepareStatement(sql)) {
    for (int i = 0; i < values.size(); i++) {
      stmt.setString(i+1, values.get(i));
    }
    ...
  } catch (SQLException e) {
    ...
  }
}
```

If you represent the blocked status with a boolean attribute or with an attribute that is `NULL` if the pallet is blocked, then you can use the MySQL function **IF(...) (https://dev.mysql.com/doc/refman/8.0/en/control-flow-functions.html#function_if)** to return the string 'yes' or 'no' depending on the value. This is illustrated in the follwing SQL code:

```
SELECT id, IF(blocked_bool_attr, 'yes', 'no') AS blocked
FROM pallets
```

Here, the attribute `blocked_bool_attr` is a boolean attribute, and `blocked` will be `'yes'` if the value of `blocked_bool_attr` is `TRUE`, otherwise `'no'`. It would yield similar results if the attribute represented non-blocked pallets with `NULL`.

# Endpoint `POST /api/v1/reset`

Resets the database to the following values:

- Customers (name, address)
  - Bjudkakor AB, Ystad
  - Finkakor AB, Helsingborg
  - Gästkakor AB, Hässleholm
  - Kaffebröd AB, Landskrona
  - Kalaskakor AB, Trelleborg
  - Partykakor AB, Kristianstad

    - Skånekakor AB, Perstorp
    - Småbröd AB, Malmö
- Cookies (name)
    - Almond delight
    - Amneris
    - Berliner
    - Nut cookie
    - Nut ring
    - Tango
- Ingredients (name, amount in stock, unit)
    - Bread crumbs, 500 000, g
    - Butter, 500 000, g
    - Chocolate, 500 000, g
    - Chopped almonds, 500 000, g
    - Cinnamon, 500 000, g
    - Egg whites, 500 000, ml
    - Eggs, 500 000, g
    - Fine-ground nuts, 500 000, g
    - Flour, 500 000, g
    - Ground, roasted nuts, 500 000, g
    - Icing sugar, 500 000, g
    - Marzipan, 500 000, g
    - Potato starch, 500 000, g
    - Roasted, chopped nuts, 500 000, g
    - Sodium bicarbonate, 500 000, g
    - Sugar, 500 000, g
    - Vanilla sugar, 500 000, g
    - Vanilla, 500 000, g

    - Wheat flour, 500 000, g
- Recipes (per batch of 100 cookies):
    - Almond delight (ingredient, amount):
        - Butter, 400
        - Chopped almonds, 279
        - Cinnamon, 10
        - Flour, 400
        - Sugar, 270
    - Amneris (ingredient, amount):
        - Butter, 250
        - Eggs, 250
        - Marzipan, 750
        - Potato starch, 25
        - Wheat flour, 25
    - Berliner (ingredient, amount):
        - Butter, 250
        - Chocolate, 50
        - Eggs, 50
        - Flour, 350
        - Icing sugar, 100
        - Vanilla sugar, 5
    - Nut cookie (ingredient, amount):
        - Bread crumbs, 125
        - Chocolate, 50
        - Egg whites, 350
        - Fine-ground nuts, 750
        - Ground, roasted nuts, 625
        - Sugar, 375

    - Nut ring (ingredient, amount):
        - Butter, 450
        - Flour, 450
        - Icing sugar, 190
        - Roasted, chopped nuts, 225
    - Tango (ingredient, amount):
        - Butter, 200
        - Flour, 300
        - Sodium bicarbonate, 4
        - Sugar, 250
        - Vanilla, 2
- Pallets
    - clear all values

For this endpoint, return the following JSON object:

```
{
  "status": "ok"
}
```

**When you implement this endpoint, make sure that you do not duplicate code!** Code duplication can be avoided, for example, by abstracting the functionality to methods which can be called with different parameters.

#### Clearing a table

To clear a table in SQL, you can use `TRUNCATE` as following:

```
TRUNCATE TABLE t
```

This will remove all rows from table `t` and reset the auto-increment value (if it is used).

#### Returning the created identifier

*If* you are using auto-increment values as keys (surrogate keys) for ingredients or cookies, you might want to get back the created identifier. This can be done by providing the option `Statement.RETURN_GENERATED_KEYS` when executing the update query. This allows you to use the method `getGeneratedKeys()` to get back all created keys (if the query created several) as a `ResultSet`. Example:

```
try (Statement statement = connection.createStatement()) {
  String sql = ...
```

```
  statement.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
  ResultSet rs = statement.getGeneratedKeys();
  if (rs.next()) {
    int createdId = rs.getInt(1);
    // do something with createdId - maybe return it
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

## Endpoint `POST /api/v1/pallets`

Creates a new pallet, where the cookie is specified with the query parameter `cookie`. Example:

```
POST /pallets?cookie=Amneris
```

That should give back (if the cookie exists and can be produced):

```
{
  "status": "ok",
  "id": <id>
}
```

Here, `<id>` should be replaced by the identifier of the produced pallet. In this assignment, you do not need to check that there are enough raw materials for producing the pallet, but the raw materials should be updated according to the recipe of the cookie. In MySQL, you can get the current date and time with the function `NOW()`, which can be used for the production date and time.

If the cookie does not exist, then return:

```
{
  "status": "unknown cookie"
}
```

For other errors, return:

```
{
  "status": "error"
}
```

(Normally, the cookie to be produced would be specified as a JSON object, but the assignment is simplified to avoid parsing JSON objects. Thus, the cookie to be produced is specified as a query parameter instead.)

# Manual Testing

## Testing the database class directly

You can create another class to test the database methods directly, for example:

```
// In file DatabaseMain.java
public class DatabaseTest {
  public static void main(String args[]) {
    Database db = new Database();
    db.connect();
    System.out.println(db.getCustomers(null, null));
  }
}
```

This can be useful when developing the first endpoints (with no query parameters). Note that this does not test the endpoints - only the methods in the database class (which are used by the endpoints).

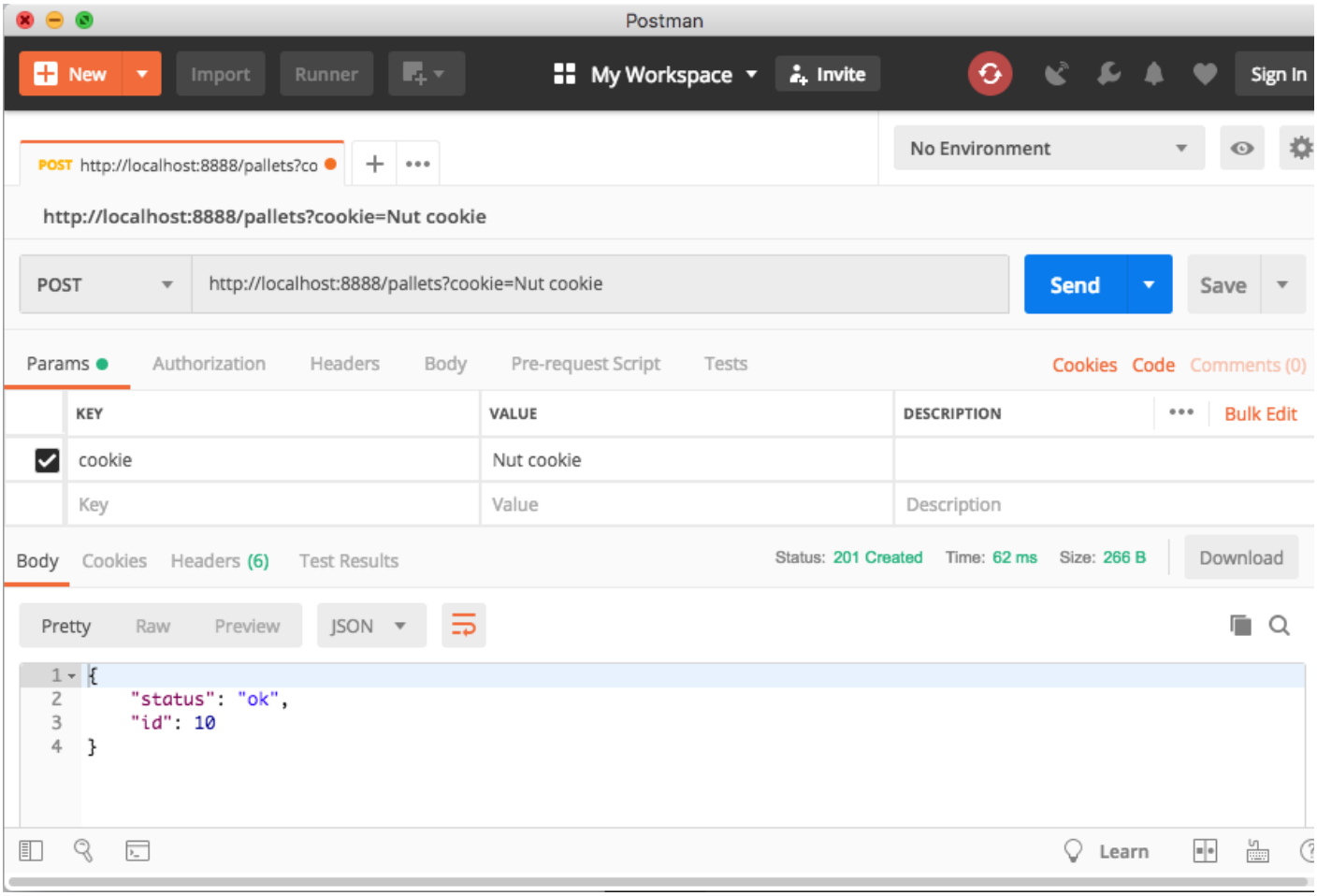
For endpoints with query parameters (`GET /pallets` and `POST /pallets`), you can create another method with the query parameters as strings and call this method instead. This is illustrated in the following code fragment:

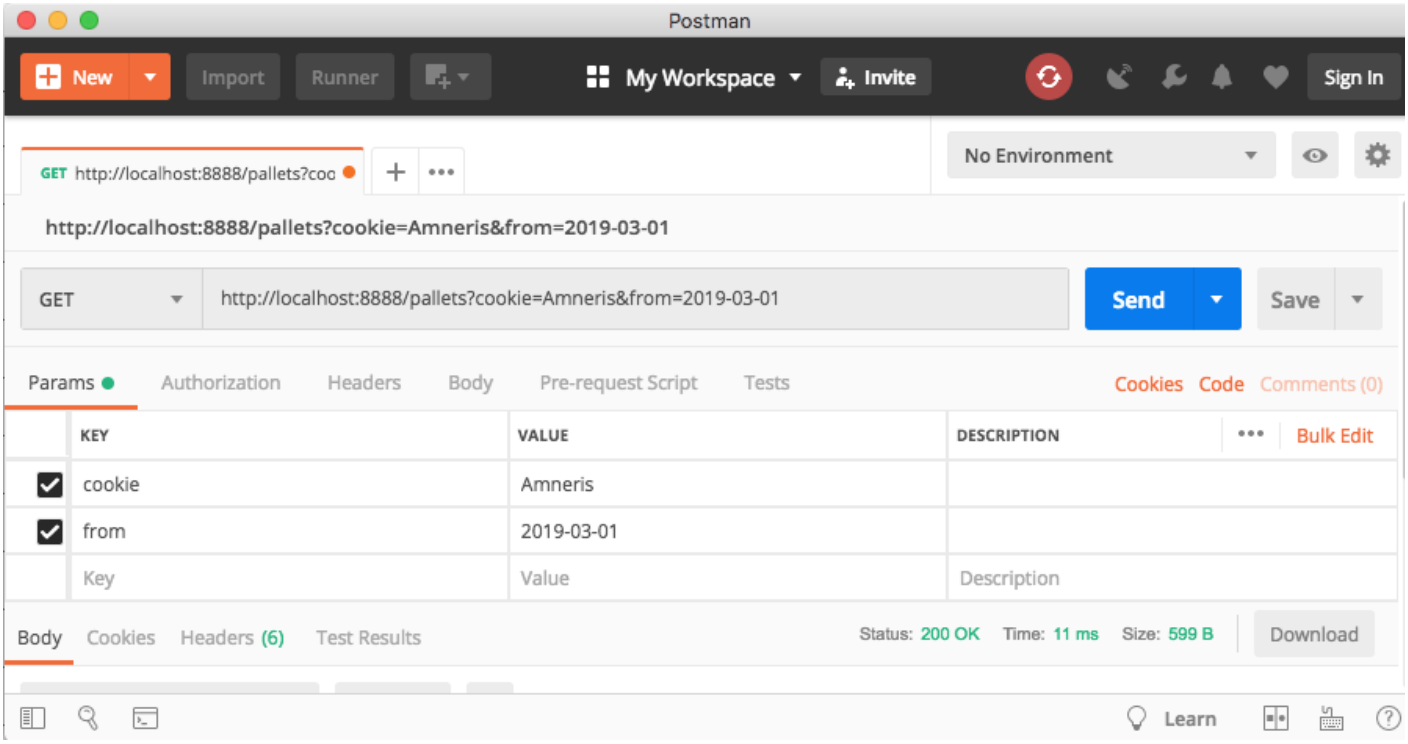
```
// In file Database.java
public class Database {
  ...

  public String createPallet(Request req, Response res) {
    if (req.queryParams("cookie") != null) {
      String cookie = req.queryParams("cookie");
      return createPallet(cookie);
    } else {
      ...
    }
  }

  protected String createPallet(String cookie) {
    ...
  }
}

// In file DatabaseTest.java
public class DatabaseTest {
  public static void main(String args[]) {
    Database db = new Database();
    db.connect();
    // Call the new method and check that it returns the expected JSON
    System.out.println(db.createCookie("Amneris"));
  }
}
```

# Testing endpoints manually with Postman

The endpoints can be tested manually using **Postman** (https://www.getpostman.com/), which is a program for testing APIs.

Using Postman, you can specify which HTTP method (GET or POST for our endpoints), the URL and query parameters. For example, to create a pallet with Nut cookies, you can specify:


Postman
New
Import
Runner
My Workspace
Invite
Sign In
POST http://localhost:8888/pallets?co
No Environment
http://localhost:8888/pallets?cookie=Nut cookie
POST
http://localhost:8888/pallets?cookie=Nut cookie
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Comments (0)
KEY
VALUE
DESCRIPTION
Bulk Edit
cookie
Nut cookie
Key
Value
Description
Body
Cookies
Headers (6)
Test Results
Status: 201 Created
Time: 62 ms
Size: 266 B
Download
Pretty
Raw
Preview
JSON
{
"status": "ok",
"id": 10
}
Learn


Postman
New
Import
Runner
My Workspace
Invite
Sign In
GET http://localhost:8888/pallets?coo
No Environment
http://localhost:8888/pallets?cookie=Amneris&from=2019-03-01
GET
http://localhost:8888/pallets?cookie=Amneris&from=2019-03-01
Send
Save
Params
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Comments (0)
KEY
VALUE
DESCRIPTION
Bulk Edit
cookie
Amneris
from
2019-03-01
Key
Value
Description
Body
Cookies
Headers (6)
Test Results
Status: 200 OK
Time: 11 ms
Size: 599 B
Download
Learn

## Testing endpoints manually with curl

It is also possible to use the terminal program `curl` to test the REST endpoints manually:

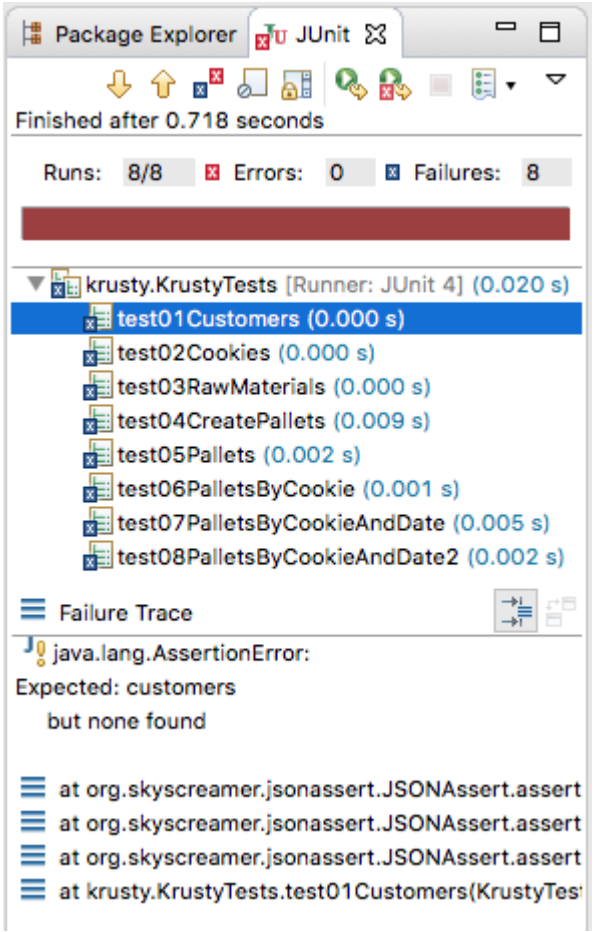
```
$ curl -X GET http://localhost:8888/api/v1/customers
```

When specifying query parameters, then spaces should be replaced with `%20`. Example:

```
$ curl -X GET http://localhost:8888/api/v1/pallets?cookie=Nut%20ring
```

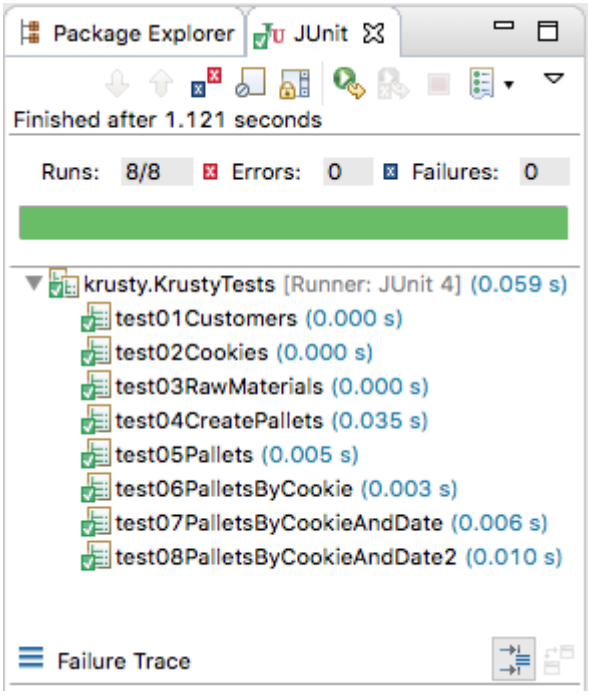
# Automatic Testing

The provided Eclipse project includes test cases written with JUnit (a testing framework) that tests the endpoints. The test class is located under `src/test/java`. Right click on the class `KrustyTests` and select `Run As -> JUnit Test`. This will run all test cases. From start, all tests will fail, as is shown in the following screenshot:




As can be seen in the bottom part of the screenshot, you can get information about why the test case failed. For the test case `test01Customers`, there is an property `customers` missing (because the database method returns an empty JSON object (`{ }`)).

When you have implemented all endpoints correctly, all tests will pass, and each test case will have a green symbol instead, as shown in the following:

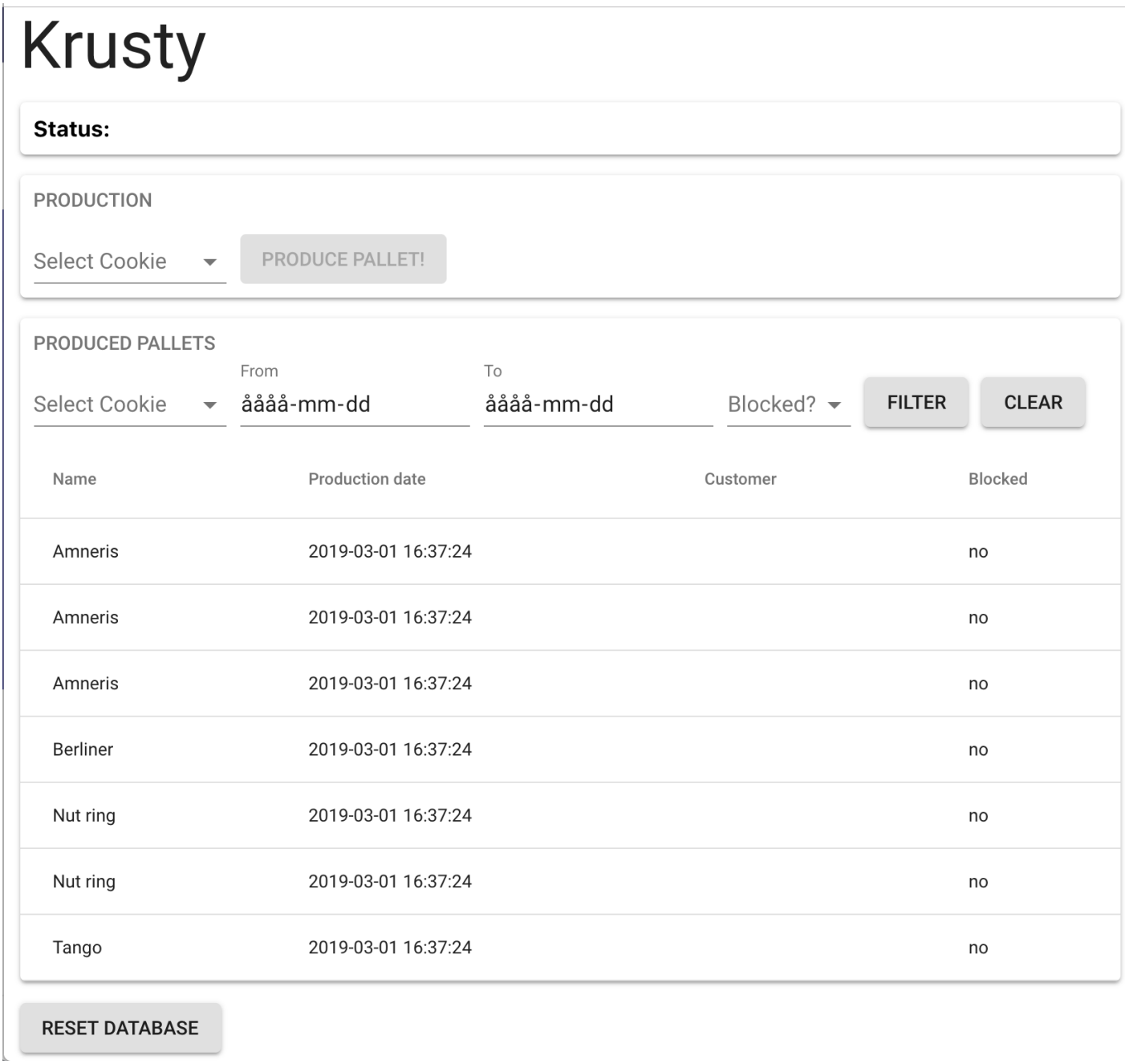



The test class will automatically start the REST server if it is not running (and stop it again afterwards if it was started). Then, before any test case is run, the test class will reset the database using the endpoint `POST /reset`. This to get back to the same state before running any test cases.

The test cases are written as methods using the testing framework JUnit. The expected output from the endspoints are stored in the directory `src/test/resources`, which you can look at if you want. These files are referenced from the test class.

# Web App

There is a web app written in **React** (https://reactjs.org/) that uses the REST API to provide a user interface, which can be accessed directly from your server by entering **http://localhost:8888/** (http://localhost:8888/) in your favorite web-browser. A backup version is provided **here** (http://fileadmin.cs.lth.se/cs/Education/edaf20/2020/project/react/) that is updated remotely and connects to your local server (that is why CORS must be accepted). When you have implemented the endpoints, the user interface will look something similar to:




The web app uses the following endpoints:

- `GET /api/v1/cookies` - for producing pallets and pallet filtering
- `GET /api/v1/pallets` - for listing pallets (with filtering if that is specified)
- `POST /api/v1/pallets` - for creating new pallets
- `POST /api/v1/reset` - for resetting the database

The web app connects to your local server (`localhost:8888`). Thus, you need to start the REST server locally if you want to use the local or remote web app (run the class `ServerMain`).

The source to the frontend is provided **here** **(https://canvas.education.lu.se/courses/10339/files/1046890/download?wrap=1)** and is based on

the **original from Niklas Fors** **(https://github.com/niklasfors/krusty-react/)** with minor changes.