**Левин В.Е., Шульга Т.Э.**

**Создание и настройка среды для работы с онтологиями используя фреймворк «Jena»**

**Учебное пособие**

# Введение

После подготовительной работы, когда Вы спроектировали схему онтологии и создали её, возникает вопрос – «Что же дальше?». Вопрос правильный, ведь для последующей работы с онтологией её надо где-то хранить, куда-то надо слать запросы на выборку, на добавление. А онтология – вещь очень гибкая и, если вы планируете расширение в будущем, могут потребоваться программы-черви, которые будут заполнять онтологию и связывать её с другими.

Средства для подобного уже существуют, и Вы способны самостоятельно создать окружение для Вашей онтологии используя язык Java и библиотеку Jena.

Созданию такого окружения, разделённого на сервер, хранящий онтологию, а также веб-интерфейс для выборки данных, и посвящено это учебное пособие.

Оглавление

[Введение 2](#_Toc443221025)

[Необходимые инструменты 4](#_Toc443221026)

[Схема разрабатываемого приложения 5](#_Toc443221027)

[Глава 1. Разработка сервера 6](#_Toc443221028)

[1.1. Создание проекта 6](#_Toc443221029)

[1.2. Подключение библиотек 9](#_Toc443221030)

[1.3. Структура проекта 12](#_Toc443221031)

[1.4. Создание окружения 15](#_Toc443221032)

[1.5. Создание модели 19](#_Toc443221033)

[Глава 2. Создание веб-интерфейса 23](#_Toc443221034)

[2.1. Создаём каркас страницы 23](#_Toc443221035)

[2.2. Подключение библиотек 23](#_Toc443221036)

[2.3. Создание логики 24](#_Toc443221037)

[2.4. Добавление разметки 26](#_Toc443221038)

[Глава 3. Запуск в эксплуатацию 28](#_Toc443221039)

# Необходимые инструменты

Для создания среды потребуется следующие инструменты:

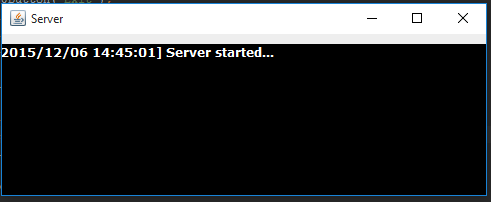
1. **Среда разработки Java приложений:** Eclipse, Intellij IDEA или NetBeans (в данном пособии используется Intellij IDEA)
2. **Runtime среда:** Java(<http://www.java.com/ru/>) и Java Development Kit (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)
3. **Фреймворк для работы с онтологиями:** для языка Java имеется Jena (<https://jena.apache.org/documentation/ontology/>)
4. **Вспомогательные библиотеки для облегчения и ускорения работы:**

* Simple-JSON (<https://github.com/fangyidong/json-simple>) – Легковесная и мощная библиотека для форматирования данных в формат JSON
* opa-core (<https://github.com/opaoz/opa-core>) – Библиотека с функциями для упрощения разворачивания клиент-серверного приложения и работы с Jena

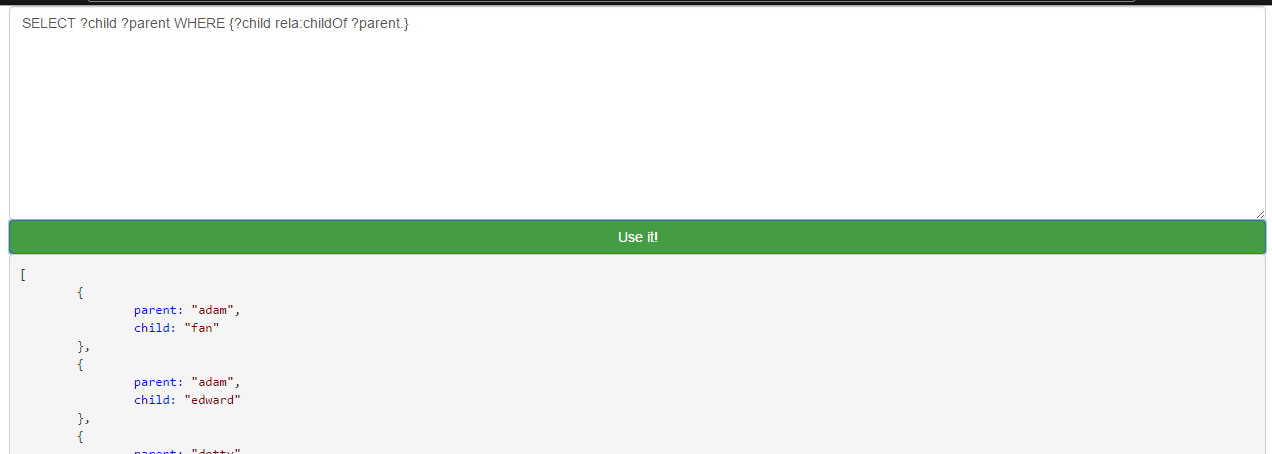
# Схема разрабатываемого приложения

Среда будет состоять из двух частей:

**Сервер**. Часть, написанная на языке Java, которая хранит в себе онтологию, обрабатывает запросы и производит выборку и добавление данных.



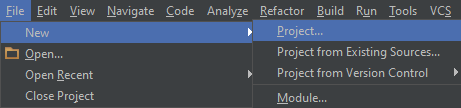
**Клиент**. Веб-интерфейс для приложения, позволяющий формировать и отсылать запросы к серверу, а также принимать результаты и выводить их в качестве наглядных данных.



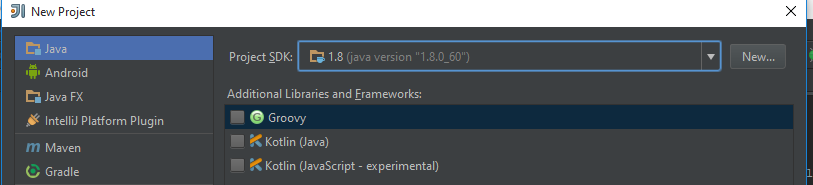
# Разработка сервера

## Создание проекта

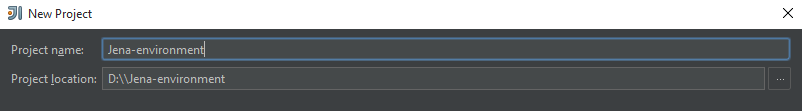
Командой меню «File > New > Project…» открываем диалоговое окно с выбором типа создаваемого проекта



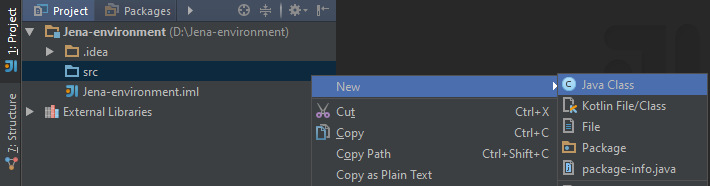
Выбираем тип проекта - «Java» и версию SDK из выпадающего списка и жмём кнопку «Next»

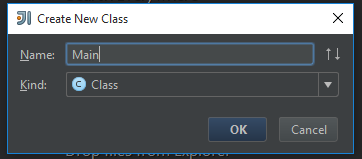


Пропускаем следующий экран с выбором темплейта, затем вводим название и расположение проекта и жмём «Finish»



Вот мы и создали проект, теперь добавим главный файл, настроим параметры запуска и выведем в консоль заветное «Hello, World!».

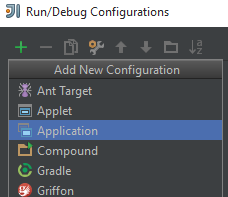




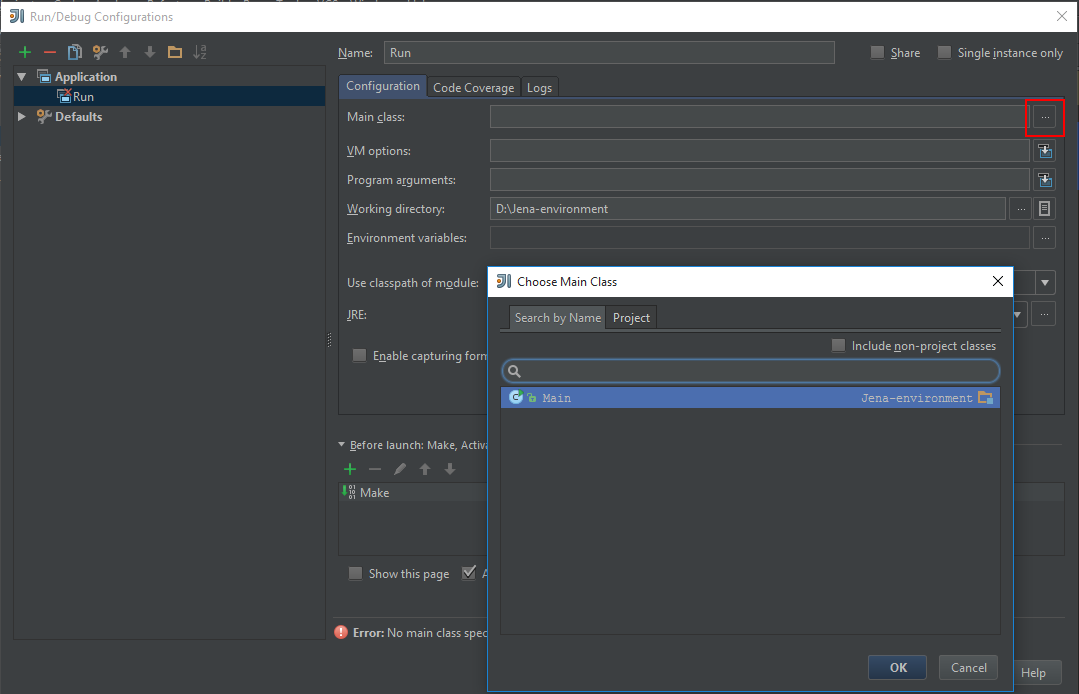
Содержимое класса:

public class Main {  
 public static void main(String[] args) {  
 System.*out*.println("Hello, world!");  
 }  
}

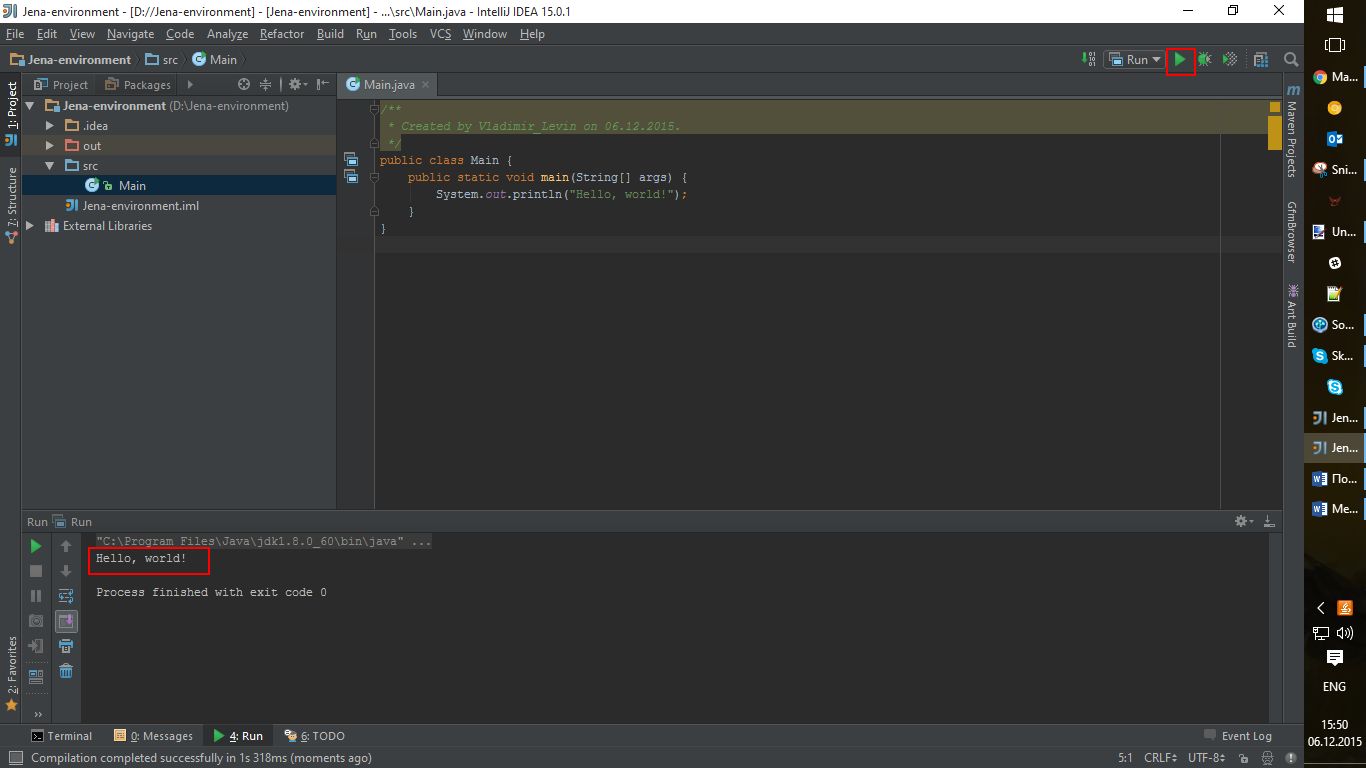
Далее настроим конфигурацию запуска. Для этого жмём выпадающее меню справа сверху и выбираем пункт «Edit Configurations…» , затем зелёный «+» и выбираем «Application»



Нажимаем «…» возле поля «Main class» и выбираем ново созданный класс Main



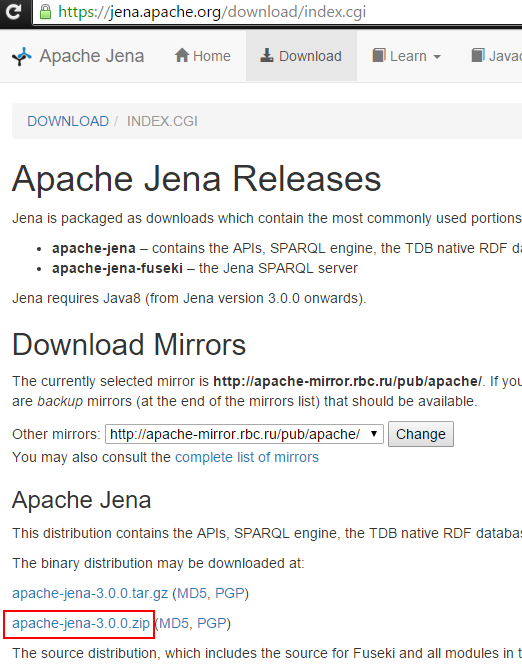
Теперь, по нажатию кнопки «Run» наш проект запустится



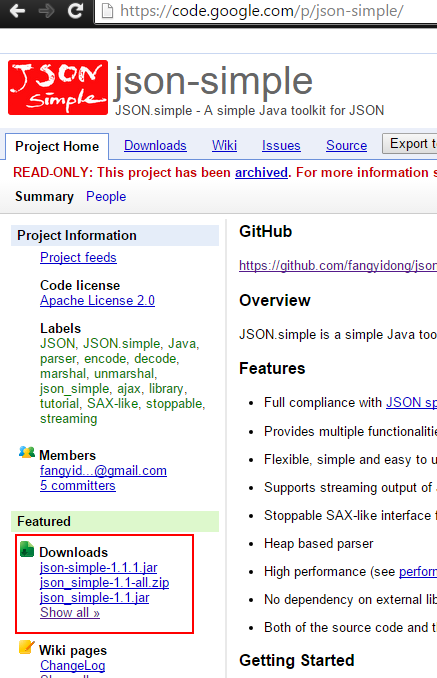
## Подключение библиотек

Скачиваем все библиотеки:

1. Jena - <https://jena.apache.org/download/index.cgi>

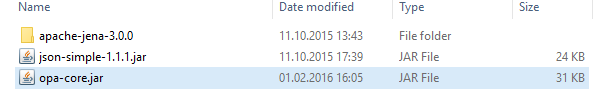


1. Simple-JSON - <https://code.google.com/p/json-simple/>

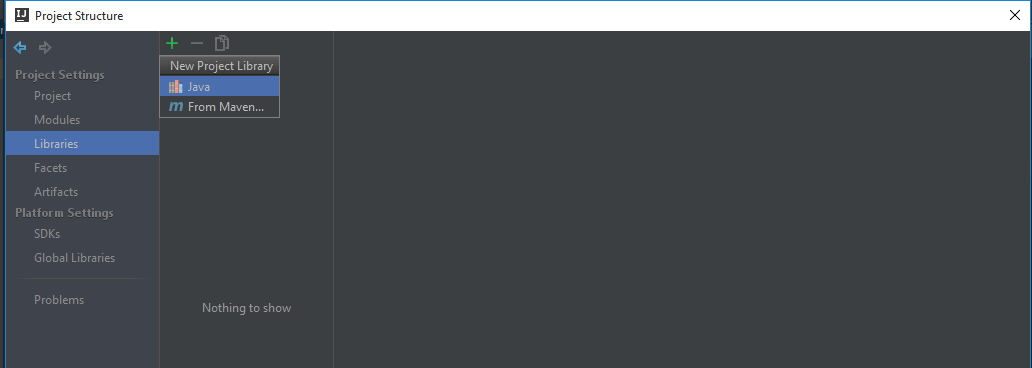


1. opa-core - <https://github.com/opaoz/opa-core/blob/master/dist/opa-core.jar?raw=true>

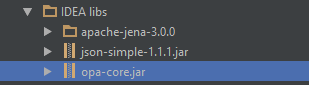
Складываем их в одну папку



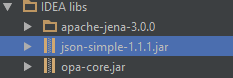
Открываем «File -> Project Structure -> Libraries»



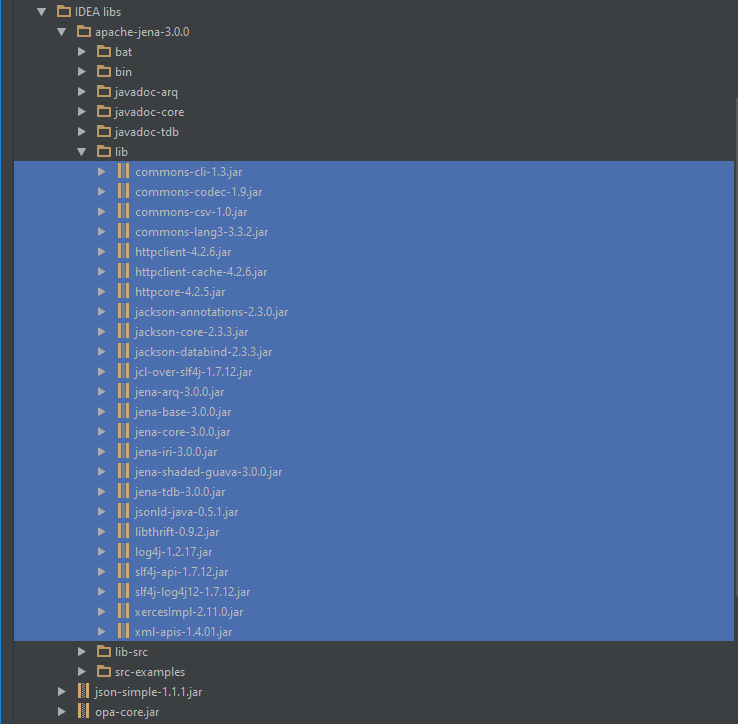
42Нажимаем на «+» и выбираем «Java», затем выбираем первый файл



Повторяем для второго



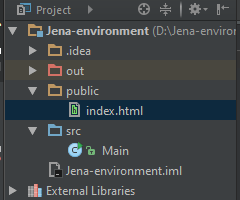
А для apache-jena выбираем всё в папке “lib”



## Структура проекта

Так как разрабатываемый проект является «клиент-серверным», возведём соответствующую структуру папок.

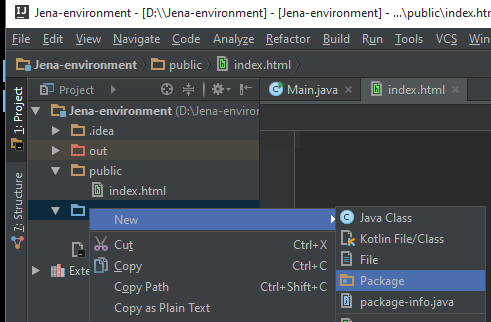
Сначала создаём папку “public” в корне проекта. Здесь будет находится вся клиентская часть проекта. Чтобы обозначить это, создадим файл “index.html”.



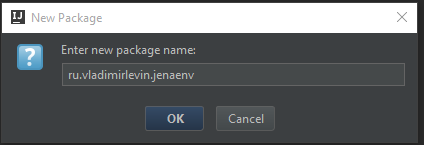
Вся серверная часть будет располагаться в папке “src”. Внутри будет возводится структура из пэкэджей (*[Package (Java)](https://ru.wikipedia.org/wiki/Package_(Java)))* в соответствии с популярным [стайлгайдом](https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html). Суть его состоит в том, что имя корневого пакета составляется из 3 частей:

1. **Домен** (org, ru, com, ua)
2. **Автор**
3. **Название проекта**

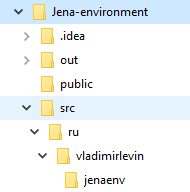
В итоге получаем *«ru.vladimirlevin.jenaenv»*



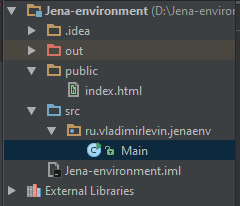
При создании можно писать наименование сразу с «.»



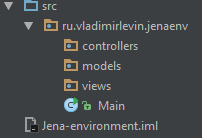
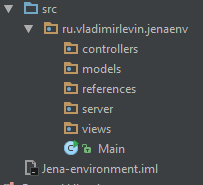
В итоге создастся соответствующая структура папок:



В созданный package и перенесём наш класс “Main”



Продолжим структурирование проекта, добавим MVC составляющую и пару вспомогательных пакетов

Суть такого подхода состоит в следующем:

1. **Controllers** – папка, содержащая логику конкретных запросов к серверу.
2. **Models** – хранилище RDF-моделей
3. **References** – папка с константами, используемыми в проекте
4. **Server** – двигатель приложения, обработка всех входящих запросов и подключений
5. **Views** – UI-компоненты, для визуализации действий сервера.
6. **Main**.java – стартер проекта

## Создание окружения

Начнём с файла конфигурации. Создадим класс **References.java** в пэкэдже «references», и добавим в него нужные константы.

Содержимое файла:

package ru.vladimirlevin.jenaenv.references;  
  
import java.awt.\*;  
import java.text.DateFormat;  
import java.text.SimpleDateFormat;  
  
public final class References {  
 /\*Model\*/  
 public static final String *FAMILY\_PATH* = "D:/Family/family.rdf"; // Путь сохранения rdf-модели  
  
 /\*Server\*/  
 public static final int *DEFAULT\_SERVER\_PORT* = 8080; // Порт по умолчанию для сервера  
 public static final int *SERVER\_PORT* = 8080; // Порт для сервера (если не указал используются дефолтный)  
  
 /\*Log\*/  
 public static final Font *LOG\_FONT* = new Font("Verdana", Font.*BOLD*, 12); // Стиль текста в консоли сервера  
 public static final DateFormat *DATE\_FORMAT* = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss"); // Формат даты в консоли  
}

Традиционно сервер запускается из-под какой-либо оболочки, так что проблемы с логами не возникает. Мы же создадим свой собственный интерфейс, для логов сервера.

Для начала добавим компонент **Log.java** (ru.vladimirlevin.jenaenv.views.components) и унаследуем его от класса *JPanel*. Log - это часть интерфейса, наглядно демонстрирующая что сейчас происходит с сервером и стилизованная под консоль.

package ru.vladimirlevin.jenaenv.views.components;  
  
import ru.vladimirlevin.jenaenv.references.References;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.text.DateFormat;  
import java.util.Date;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class Log extends JPanel {  
 private static final long *serialVersionUID* = -7968405633747657642L; // Автоматически генерируемый UID  
 private JTextArea text; // Тестовое представление логов  
 private DateFormat dateFormat = References.*DATE\_FORMAT*; // Паттерн форматирования даты из файла References  
  
 // Конструктор класса  
 public Log() {  
 text = new JTextArea(); // Добавляем текст на панель  
 text.setAlignmentX(*LEFT\_ALIGNMENT*); // Позиционируем её налево  
 text.setAlignmentY(*TOP\_ALIGNMENT*); // и прижимаем к верху  
 text.setBackground(Color.*BLACK*); // Задний фон - чёрный  
 text.setForeground(Color.*WHITE*); // Цвет текста - белый  
 text.setFont(References.*LOG\_FONT*); // Стиль шрифта из файла References  
 text.setLineWrap(true); // Устанавливаем перенос строк  
 text.setWrapStyleWord(true); // И слов  
  
 JScrollPane scrollPane = new JScrollPane(text); // Создаём ScrollBar  
 scrollPane.setPreferredSize(new Dimension(500, 200)); // Ограничиваем размеры области вывода  
 this.add(scrollPane); // Добавляем к нашей импровизированной консоли  
 this.setVisible(true); // Отрисовываем  
 }  
  
 // Метод добавления сообщений в консоль  
 public void info(String message) {  
 text.append("[" + dateFormat.format(new Date()) + "] " + message + " \n");  
 }  
  
 // Размеры окна консоли  
 @Override  
 public Dimension getPreferredSize() {  
 return new Dimension(500, 200);  
 }  
}

После добавления компонента, нам требуется класс, который будет непосредственно показывать окно. Создадим для этого класс **ServerUI.java** (ru.vladimirlevin.jenaenv.views) и унаследуем его от класса *JFrame*

package ru.vladimirlevin.jenaenv.views;  
  
import ru.vladimirlevin.jenaenv.views.components.Log;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class ServerUI extends JFrame {  
 private static final long *serialVersionUID* = 6182239340196405650L;  
 private Log log; // Окно лога, созданное в предыдущем шаге  
  
 public ServerUI() {  
 log = new Log(); //Инициализируем окно  
  
 this.setLayout(new FlowLayout()); // Т.к. у нас всего один компонент в окне, установим данный тип заполнения  
 this.add(log); // Добавляем окно лога на фрейм  
 this.setTitle("Server"); // Устанавливаем название  
 this.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*); // При закрытии окна, выключаем сервер  
 this.pack(); // Устанавливаем размер по контенту  
 this.setVisible(true); // Отрисовываем  
 }  
  
 // Метод добавления сообщения в лог  
 public void println(String message) {  
 log.info(message);  
 }  
}

Т.к. сервер будет принимать Sparql запросы и обрабатывать их, создадим контроллер **RequestManager.java** (ru.vladimirlevin.jenaenv.controllers). Оставим его пока пустым, до создания модели.

package ru.vladimirlevin.jenaenv.controllers;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class RequestManager {  
 public RequestManager(String request) {  
 // Обаратываем запрос  
 }  
}

Обработку подключений на сервере следует осуществлять в отдельном потоке, создадим для этого класс **ClientThread.java** (ru.vladimirlevin.jenaenv.server), и реализуем в нём интерфейс *Runnable*

package ru.vladimirlevin.jenaenv.server;  
  
import ru.opa.pack.net.HttpResponse;  
import ru.vladimirlevin.jenaenv.controllers.RequestManager;  
import ru.vladimirlevin.jenaenv.views.ServerUI;  
  
import java.io.\*;  
import java.net.Socket;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class ClientThread implements Runnable {  
 private Socket socket; // Подлючённый юзер  
 private InputStream inputStream; // Сообщения от юзера  
 private OutputStream outputStream; // Ответ юзеру  
 private String request; // Запрос от юзера  
 private ServerUI ui; // Окно консоли  
  
 // Инициализируем новый поток  
 protected ClientThread(Socket socket, ServerUI ui) throws IOException {  
 this.socket = socket;  
 this.inputStream = socket.getInputStream();  
 this.outputStream = socket.getOutputStream();  
 this.ui = ui;  
 new Thread(this).start(); // Запускаем обработку в другом потоке  
 }  
  
 // Эта функция вызовется автоматически при запуске потока  
 public void run() {  
 try {  
 BufferedReader in = new BufferedReader(new InputStreamReader(inputStream));  
 String line = in.readLine(); // Считываем первую линию запроса  
 if (line.startsWith("POST")) {  
 request = HttpResponse.*readBodyPOST*(in, line); // Если это ПОСТ-запрос, считываем тело запроса  
 if (!request.equals("none")) { // Если тело запроса не пустое, передаём его нашему RequestManager'у  
 // и отправляем ответ назад  
 HttpResponse.*writeResponse*(new RequestManager(request).toString(), outputStream);  
 }  
 } else if (line.startsWith("GET")) { // Ести это гет запрос  
 HttpResponse.*sendFileGET*(outputStream, line); // отправляем файл в ответ на гет запрос  
 }  
 } catch (IOException t) { // Обратока ошибок  
 ui.println("Socket error");  
 ui.println(t.toString());  
 } finally {  
 try {  
 socket.close(); // Закрываем полток после ошибки  
 } catch (Throwable t) {  
 ui.println("Socket closing error");  
 }  
 }  
 ui.println("Client processing finished");  
 }  
}

После создания обработчиков, нужно запустить сервер, который будет распределять запросы. Файл **Server.java** (ru.vladimirlevin.jenaenv.server)

package ru.vladimirlevin.jenaenv.server;  
  
import ru.vladimirlevin.jenaenv.references.References;  
import ru.vladimirlevin.jenaenv.views.ServerUI;  
  
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class Server {  
 private ServerSocket serverSocket; // Объект сервера, слушающий подключения  
 private ServerUI ui; // Визуальное представление консоли  
  
 public Server(int port, ServerUI ui) throws IOException {  
 if (port < 0 || port > 65535) { // Если передан неверный порт  
 port = References.*DEFAULT\_SERVER\_PORT*; // устанавливаем дефотный  
 ui.println("Server port changed to default value - " + port);  
 }  
  
 this.ui = ui;  
 serverSocket = new ServerSocket(port); // Инициализируем объект сервера  
 ui.println("Server started...");  
 }  
  
 public void start() throws IOException {  
 while (true) { // Бесконечный цикл ожидания подключений  
 Socket socket = serverSocket.accept(); // Как только клиен присоединился  
 ui.println("Client connected with " + socket.getLocalSocketAddress());  
 new ClientThread(socket, ui); // Выделяем ему поток  
 }  
 }  
}

После того как мы создали окружение, отредактируем файл **Main.java**, проинициализировав в нём сервер.

package ru.vladimirlevin.jenaenv;  
  
import ru.vladimirlevin.jenaenv.references.References;  
import ru.vladimirlevin.jenaenv.server.Server;  
import ru.vladimirlevin.jenaenv.views.ServerUI;  
  
import java.io.IOException;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 06.12.2015.  
 \*/*public class Main {  
 public static ServerUI *ui* = new ServerUI();  
  
 public static void main(String[] args) {  
 try {  
 *ui*.setVisible(true);  
 new Server(References.*SERVER\_PORT*, *ui*).start();  
 } catch (IOException e) {  
 *ui*.println("Server crashed" + e);  
 }  
 }  
}

## Создание модели

Теперь приступим к созданию RDF-модели. Для примера создадим онтологию «Семейное Древо». Создадим класс **FamilyTree.java** (ru.vladimirlevin.jenaenv.models)

package ru.vladimirlevin.jenaenv.models;  
  
import org.apache.jena.rdf.model.\*;  
import org.apache.jena.util.FileManager;  
import ru.vladimirlevin.jenaenv.references.References;  
  
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.util.HashMap;  
import java.util.Map;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class FamilyTree {  
 public static final String *FAMILY\_URI* = "http://family/"; // Префикс для модели  
 public static final String *RELATIONSHIP\_URI* = "http://purl.org/vocab/relationship/"; // Префикс для связей в модели  
 private Map<String, Resource> family = new HashMap<String, Resource>(); // Двумерный массив объектов  
 private Model model; // Модель  
  
 public FamilyTree() {  
 try {  
 // Если файл модели уже существует, загружаем его  
 model = FileManager.*get*().loadModel(References.*FAMILY\_PATH*);  
 } catch (Exception e) {  
 // Если файл не найден, создаём модель и экспортируем её  
 model = reCreateModel();  
 *exportModel*(model);  
 }  
 }  
  
 public Model reCreateModel() {  
 model = ModelFactory.*createDefaultModel*(); // Создаём дефолтную модель  
 Resource NAMESPACE = model.createResource(*RELATIONSHIP\_URI*); // Создаём NAMESPACE модели  
 model.setNsPrefix("rela", *RELATIONSHIP\_URI*); // Устанавливаем префик для отношений  
  
 Property childOf = model.createProperty(*RELATIONSHIP\_URI*, "childOf"); // Отношение "ребёнок"  
 Property siblingOf = model.createProperty(*RELATIONSHIP\_URI*, "siblingOf"); // Отношение "брат"  
 Property spouseOf = model.createProperty(*RELATIONSHIP\_URI*, "spouseOf"); // Отношение "супруг"  
 Property parentOf = model.createProperty(*RELATIONSHIP\_URI*, "parentOf"); // Отношение "родитель"  
  
 // Объекты людей  
 family.put("Adam", model.createResource(*FAMILY\_URI* + "adam"));  
 family.put("Dotty", model.createResource(*FAMILY\_URI* + "dotty"));  
 family.put("Beth", model.createResource(*FAMILY\_URI* + "beth"));  
 family.put("Chuck", model.createResource(*FAMILY\_URI* + "chuck"));  
 family.put("Edward", model.createResource(*FAMILY\_URI* + "edward"));  
 family.put("Fan", model.createResource(*FAMILY\_URI* + "fan"));  
 family.put("Greg", model.createResource(*FAMILY\_URI* + "greg"));  
 family.put("Harriet", model.createResource(*FAMILY\_URI* + "harriet"));  
  
 // Отношения между ними  
 push("Adam", spouseOf, "Dotty");  
 push("Beth", spouseOf, "Chuck");  
 push("Fan", spouseOf, "Greg");  
  
 push("Adam", siblingOf, "Beth");  
 push("Edward", siblingOf, "Fan");  
  
 push("Edward", childOf, "Adam");  
 push("Edward", childOf, "Dotty");  
 push("Fan", childOf, "Adam");  
 push("Fan", childOf, "Dotty");  
 push("Harriet", childOf, "Fan");  
 push("Harriet", childOf, "Greg");  
  
 push("Adam", parentOf, "Edward");  
 push("Dotty", parentOf, "Edward");  
 push("Adam", parentOf, "Fan");  
 push("Dotty", parentOf, "Fan");  
 push("Fan", parentOf, "Harriet");  
 push("Greg", parentOf, "Harriet");  
  
 // После создания возвращаем модель  
 return model;  
 }  
  
 // Метод установки связи между объектами  
 public void push(String who, Property what, String whom) {  
 model.add(model.createStatement(family.get(who), what, family.get(whom)));  
 }  
  
 // Вспомошательный метод получения модели  
 public Model getModel() {  
 return model;  
 }  
  
 // Экспорт модели  
 public static Boolean exportModel(Model model) {  
 File file = new File(References.*FAMILY\_PATH*); // Создание файла  
  
 try {  
 file.createNewFile();  
 } catch (IOException e) {  
 return false;  
 }  
 // Вывод в файл  
 try (FileOutputStream fout = new FileOutputStream(file)) {  
 RDFWriter writer = model.getWriter("RDF/XML-ABBREV");  
 writer.write(model, fout, null);  
 } catch (IOException e) {  
 return false;  
 }  
  
 return true;  
 }  
}

После создания модели, отредактируем файл **RequestManager.java**, так, чтобы в нём использовалась наша модель.

package ru.vladimirlevin.jenaenv.controllers;  
  
import org.apache.jena.rdf.model.Model;  
import ru.opa.pack.util.JSONHelper;  
import ru.opa.pack.util.QueryExec;  
import ru.vladimirlevin.jenaenv.models.FamilyTree;  
  
*/\*\*  
 \* Created by Vladimir\_Levin on 14.02.2016.  
 \*/*public class RequestManager {  
 private String jsonString;  
  
 private static FamilyTree *familyTree* = new FamilyTree(); // Загружаем  
 private static Model *model* = *familyTree*.getModel(); // И получаем модель  
  
 public RequestManager(String request) {  
 // Выполняем SparQL запрос и возвращаем ответ  
 jsonString = JSONHelper.*generateJSONResponse*(QueryExec.*exec*(request, *model*));  
 }  
  
 @Override  
 public String toString() {  
 return jsonString;  
 }  
}

Вот и всё. На это создание сервера законченно. Приступим к созданию интерфейса приложения.

# Создание веб-интерфейса

## Создаём каркас страницы

Открываем файл **public/index.html** и создаём стандартную структуру страницы.

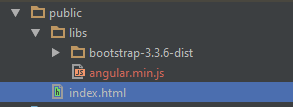
<!doctype html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Document</title>  
</head>  
<body>  
  
</body>  
</html>

## Подключение библиотек

Потребуются следующие библиотеки:

1. **Bootstrap** – библиотека стилей для страниц. Позволяет быстро создать интерфейс, понятный и привычный юзеру. Имеет большое количество предустановленных стилей (<http://getbootstrap.com/getting-started/)>
2. **Angular** – JavaScript фреймворк имеющий уйму применений во всех областях ( <https://angularjs.org/>)

Скачиваем и складываем всё в папку *libs*

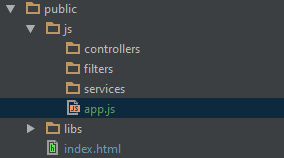


Подключаем в файле **index.html**

<!doctype html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Map</title>  
 <link rel="stylesheet" href="libs/bootstrap-3.3.6-dist/css/bootstrap.css">  
</head>  
<body>  
  
<script src="libs/angular.min.js"></script>  
</body>  
</html>

## Создание логики

Создадим папку “js” рядом с файлом **index.html** и возведём в ней следующую структуру:



1. **Controllers –** в этой папке будет храниться основной обработчик взаимодействия с пользователем
2. **Filters –** здесь будут лежать модули, необходимые для форматирования приходящей даты
3. **Services –** непосредственно сервисы, осуществляющие запросы к бекенду
4. **app.js –** основной файл, регистрирующей все основные в приложении

Начнём создание логики с файла **app.js**

'use strict';  
 angular  
 .module('App', []) // Инициализация модуля  
 .constant('config', { // Иницциализация константы  
 queryUrl: '/'  
 });

Создадим файл **http-request.js** (public/js/services), который будет осуществлять запросы к серверу

'use strict';  
  
angular  
 .module('App')  
 .service('httpRequest', ['$http', 'config', function ($http, config) { //Создание сервиса  
 this.send = function (message) { // Метод сервиса  
 return $http({ // Отправка запроса  
 url: config.queryUrl, // URL из конфига  
 method: 'POST', // Метод ПОСТ  
 headers: { // Хедеры для Jena  
 'Content-Type': 'application/x-www-form-urlencoded',  
 },  
 data: message // Сообщение  
 });  
 }  
 }]);

Далее создадим файл **main.js** (public/js/controllers), который будет отвечать за взаимодействие с пользователем.

'use strict';  
  
angular  
 .module('App')  
 .controller('MainController', ['httpRequest', function (httpRequest) { // Создание контроллера  
 var vm = this;  
  
 vm.\_init\_ = function () { // Инициализация  
 vm.query = { // Пример запрос  
 title: 'Parent', // Название  
 text: 'SELECT ?child ?parent WHERE {?child rela:childOf ?parent.}' // Sparql запрос  
 };  
 vm.output = 'Text will be here'; // Переменная вывода  
 };  
  
 vm.send = function (data) { // Функция отправки запроса  
 httpRequest.send('PREFIX rela: <http://purl.org/vocab/relationship/>' + // Объявление префикса  
 data.text) // Добавление запроса  
 .then(function (response) {  
 console.log(response);  
 vm.output = response.data; // Присваивание вывода  
 });  
 };  
 }]);

Для форматирования вывода создадим пару фильтров:

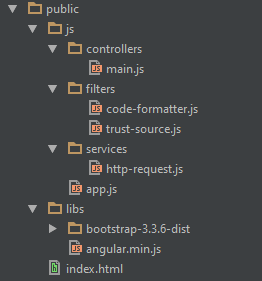
**code-formatter.js** (public/js/filters)

'use strict';  
  
angular  
 .module('App')  
 .filter('codeFormatter', function () {  
 return function (value) { // Функция фильтра  
 if (angular.isObject(value) || angular.isArray(value)) {  
 value = angular.toJson(value); // Если пришёль не JSON превращаем в JSON  
 }  
  
 if (!angular.isString(value)) {  
 return value; // Если значение некорректно - возвращаем  
 }  
  
 return value.replace(/(\[|,|\{)/g, '$1\n') // Делаем строку мультилайновой  
 .replace(/]/g, '\n]') // Добавляем переносы  
 .replace(/"([a-z]+)":/g, '\t\t<code style="color:blue;">$1: </code>') // Перекрашиваем имя поля в синий  
 .replace(/"http:\/\/family\/([a-z]+)"/g, '<code style="color:maroon">"$1"</code>') // Перекрашивае значение поля в красный  
 .replace(/\{/g, '\t{') // Добавляем табуляции для открывающих скобок  
 .replace(/}/g, '\n\t}'); // Добавляем табуляции для закрывающих скобок  
 };  
 });

**trust-source.js** (public/js/filters)

'use strict';  
  
angular  
 .module('App')  
 .filter('trust', function ($sce) {  
 return function (value, source) { // Функция фильтра  
 var result;  
 switch (source.toLowerCase()) {  
 case 'url':  
 result = $sce.trustAsResourceUrl(value); // Ресурс как URL  
 break;  
 case 'html':  
 result = $sce.trustAsHtml(value); // Ресурс как HtmL разметка  
 break;  
 case 'js':  
 $sce.trustAsJs(value); // Ресурс как JS код  
 break;  
 default:  
 result = value;  
 }  
 return result;  
 };  
 });

В финале у нас должна получиться следующая структура файлов:



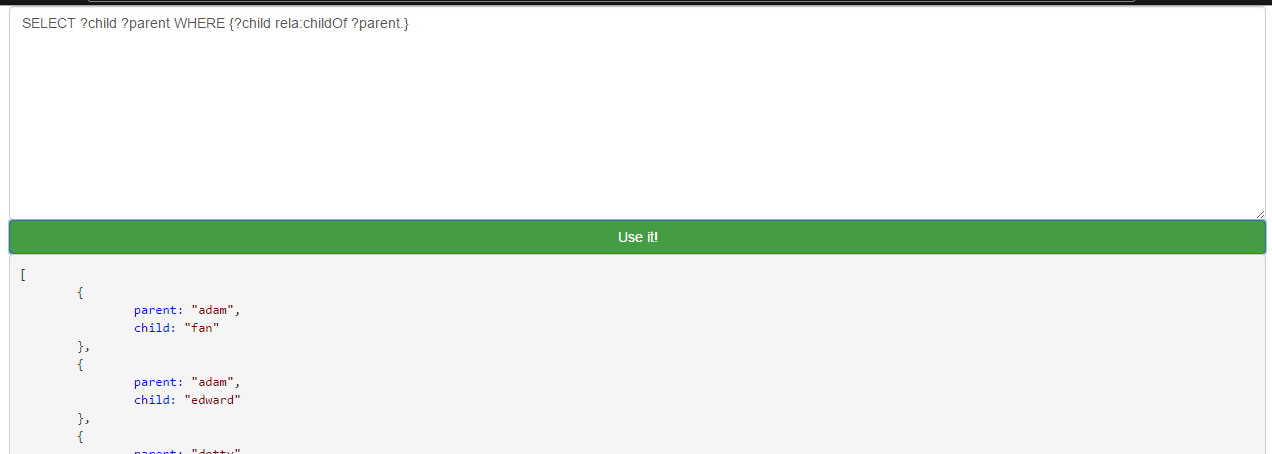
## Добавление разметки

Отредактируем **index.html**, подключив в нём наши скрипты и добавив разметки

<!doctype html>  
<html lang="en" ng-app="App">  
<head>  
 <meta charset="UTF-8">  
 <title>Map</title>  
 <link rel="stylesheet" href="libs/bootstrap-3.3.6-dist/css/bootstrap.css">  
</head>  
<body ng-controller="MainController as vm" ng-init="vm.\_init\_()">  
<div class="container-fluid" ng-init="vm.\_init\_()">  
 <div class="row">  
 <div class="col-xs-12 text-center">  
 <strong>{{vm.query.title}}</strong>  
 <textarea ng-model="vm.query.text" cols="30" rows="10" class="form-control"></textarea>  
 <button class="btn btn-success btn-block" ng-click="vm.send(vm.query)">Use it!</button>  
 </div>  
 </div>  
 <div class="row output">  
 <div class="col-xs-12">  
 <pre ng-bind-html="vm.output | codeFormatter | trust : 'html'"></pre>  
 </div>  
 </div>  
</div>  
  
<script src="libs/angular.min.js"></script>  
  
<script src="js/app.js"></script>  
<!--Controllers-->  
<script src="js/controllers/main.js"></script>  
  
<!--Services-->  
<script src="js/services/http-request.js"></script>  
  
<!--Filters-->  
<script src="js/filters/code-formatter.js"></script>  
<script src="js/filters/trust-source.js"></script>  
</body>  
</html>

# Запуск в эксплуатацию

Теперь, когда и фронтенд и бекенд готовы, можно стартовать приложение, открыть в браузере url [*http://127.0.0.1:8080/*](http://127.0.0.1:8080/)и при нажатии на кнопку “Use It!” приложение отправит запрос из поля ввода на сервер и вернёт результат.



Также Вы можете писать туда свои запросы и проверять, какой результат придёт по ним.

Все файлы проекта доступны в [Git-репозитории](https://github.com/opaoz/JenaEnvironment)