Geekbrains

**Система автоматизированного расчета фонда обеспечения складских запасов на предприятиях розничной и оптовой торговли.**

Программа: Разработчик - Веб- разработка на Java

Специализация: Программист

ФИО: Давыдов Николай Геннадьевич

Город Москва

Год 2024

**Содержание**

1. Введение – 3 стр.

2. Разработка проекта - 5 стр.

2.1. Основная вычислительная логика проекта – 5 стр.

2.2. Разработка пользовательского интерфейса – 17 стр.

3. Инструкция по работе с программой – 29 стр.

4. Выводы, планы по доработке и развитию проекта - 37 стр.

5. Список используемых инструментов и литературы. – 38 стр.

**Введение**

Имея длительный опыт работы менеджером ВЭД / начальником отдела закупок и логистики, я часто сталкивался с проблемой отсутствия необходимых оборотных средств на закупки товара для своевременное пополнения складских запасов. Это происходит по многим причинам,

- не просчитанное расширение ассортимента продукции и введение новых поставщиков (в ущерб финансирования заказов уже устоявшегося ассортимента популярных товаров), чрезмерное снятие владельцем компании «сливок» с бизнеса на свои нужды, недостаточное финансирование закупок каких-то наиболее критичных, популярных товаров и т.д. Лично сталкивался со всеми этими случаями. В корне этих причин лежат ситуативные решения и отсутствие правильного планирования и расчета бюджета, как доходной части, так и расходной, в том числе необходимого на пополнение склада. Часто бизнес перерастает из небольшого проекта в более серьезное предприятие со складами и офисами, и прочей инфраструктурой, а мышление остается на прежнем уровне, или нет понимания, что многие вещи можно спланировать, предугадать и рассчитать с той или иной долей вероятности.

Большинство торговых компаний в нашей стране используют программу «1С управление торговлей» для торгового и складского учета. В целом, лично я и наша торговая компания имеем не слишком положительный опыт работы с компаниями, производящими обслуживание и настройку 1С. Просьбы по настройке отчетов производятся медленно, дорого, и бывает, что результаты этих отчетов вызывают сомнения. Алгоритмы расчетов непрозрачны, при ручной проверке бывает, что один и тот же отчет в каких-то случаях считает верно, в каких-то совсем нет. Да и программисты, не зная специфики работы на твоей позиции в твоей компании, могут не до конца понимать, что от них требуется. Также имеем опыт общения с компаниями, которые предлагают свои программы по ведению закупок. Они исходят из своих парадигм и представлений. Например, одна компания исходила из того, что запасы должны в любом случае пополняться своевременно и так часто, как только возникает необходимость, невзирая на объем партий товара. Для нас это совершенно не подходит, так-как по нашему многолетнему опыту, у разных поставщиков из разных стран есть какая-то минимальная партия, меньше которой товар везти просто невыгодно. Такой подход бы нас разорил за год. В то же время, для какой-то другой компании такой подход был бы идеальным.

В результате я решил написать программу для расчета бюджета и заказов поставщикам со своей бизнес-логикой и отразить то, как я вижу решение этой задачи изнутри, опираясь на свой собственный опыт.

Думаю, он более-менее общий для небольших и средних для торговых компаний, занимающихся внешнеэкономической деятельностью.

**2. Разработка проекта**

**2.1. Основная вычислительная логика проекта**

Проект я начал делать еще до изучения фреймворка Spring, в “Hello world” формате. Задачей на этом этапе было написать основную логику и расчеты. Общая идея проекта – подсчет прогнозируемого бюджета на закупки и доставку товара через расчет прогнозируемых заказов товара в течение расчетного периода. Таким образом у пользователя должен появиться понедельный план расходов на возобновление складских остатков, который помогает при планировании общего бюджета компании и принятии управленческих решений, в том числе и в области закупок товара.

Сами расчеты находится в папке **calculation**, а классы, описывающие сущности (поставщиков, товары и прочее) в папке **models**.

Вместо базы данных на начальном этапе были использованы и искусственно проинициализированы коллекции List для

- списка производителей (поставщиков) товаров – (шаблон для поставщика - класс Supplier)

- списка товаров для каждого производителя (шаблон для товара – класс Item) . У каждого поставщика есть список товаров – поле List<Item> items.

- для каждого товара Item также необходимо заполнить поле

List<Double> **salesPerWeek –** список предполагаемого количества проданного товара в каждую неделю расчетного периода. Специалисты по закупкам всегда ориентируются при расчете заказов на статистику продаж/потребления предыдущих периодов. Расчетный период по умолчанию в моей программе установлен 52 недели (1 год) – поле *weeksNum* класса Parameters . В каждом элементе i-том salesPerWeek содержится количество проданного товара за i-тую неделю расчетного периода. На начальном этапе я его заполнял искусственно, так было легче и для проверки правильности расчетов. (В окончательном варианте данные считываются их специально подготовленного excel файла, данные для которого выгружаются из простого отчета программы 1С, об этом более детально распишу далее).

Для объяснения логики работы программы, рассмотрим упомянутые классы Supplier, Item, Paramenetrs более подробно

**Класс Supplier**

В классе Supplier – я постарался создать необходимые поля, характеризующие поставщика.

- @Id  
 **private long id**; - id поставщика

@Column(name = **"name"**)

- **private** String **name**; - наименование поставщика

@Column(name = **"min\_order\_amount"**)

**private double minOrderAmount**; - это поле показывает минимальную величину заказа в валюте поставщика. При минимальном опыте работы становится понятно, что меньше какой-то определенной суммы везти товар становится нерентабельно из-за сопутствующих расходов на транспортировку, склады временного хранения, документальное сопровождение внешнеэкономического контракта и таможенную очистку. Для каждого поставщика это индивидуально, в зависимости от страны и типа товара.

@Column(name = **"lead\_time"**)  
 **private int leadTime**; *//in weeks* - это поле показывает обычное количество недель, необходимое поставщику для изготовления и подготовки товара к отправке в наш адрес. Это тоже выясняется из опыта работы с поставщиком или запрашивается напрямую. Как правило оно плюс минус одно и тоже в течение года.@Column(name = **"delivery\_time"**)  
 **private int deliveryTime**; *//in weeks* - это поле показывает обычное количество недель, необходимое транспортной компании для доставки товара от поставщика на наш склад. Как правило оно плюс минус одно и тоже в течение года.@Column(name = **"order\_period"**)  
 **private int orderPeriod**; это поле показывает указывает на желательный промежуток между заказами в неделях, практически оно означает на какое количество недель мы планируем обеспечить свою компанию товаром при поступлении каждого нового заказа поставщика на наш склад. Это поле логически дополняет поле ***minOrderAmount****,* которое ограничивает наш заказ снизу, чтобы мы не заказывали слишком частые и маленькие нерентабельные партии. Поле **orderPeriod** ограничивает нас сверху, чтобы мы не заказывали товара слишком много**.**

К пониманию этого значения и закупщик, и руководство приходят тоже довольно быстро. Слишком частые закупки и не рентабельны, и приводят к большим нагрузкам на сопровождающие их отделы и склад. Слишком редкие, но, следовательно, слишком крупные закупки требуют чрезмерной траты оборотных средств, потери гибкости в политике закупок и накоплению неликвида и портящегося товара.

@Column(name = **"delivery\_cost\_factor"**)  
 **private double deliveryCostFactor**; коэффициент для расчета расходов на доставку и таможенную очистку груза. Можно в дальнейшем их рассчитывать более точно, если добавить для каждого товара поле таможенного кода с соответствующей пошлиной, а также вес и объем. Но в целом, для планирования бюджета можно обойтись этим ориентировочным коэффициентом. По нашему опыту работы мы всегда можем сказать заранее ориентировочную себестоимость товара с учетом доставки до склада. Для нашего поставщика химической продукции из Германии это в среднем 1,35 (бывает 1,33 - бывает 1,36 то есть колеблется незначительно) , для аналогичной продукции с юга это Италии уже 1,45 из-за большего расстояния и другой структуры продукции, может тоже от заказа к заказу колебаться от 1,43 до 1,46, то есть колебаниями можно пренебречь.

С помощью этого коэффициента мы можем рассчитать себестоимость каждой единицы продукции на нашем складе. Допустим, цена товара с завода 100 евро, себестоимость с учетом доставки и очистки можно получить, умножив эту цену на коэффициент 1,35, то есть себестоимость товара получится 135 евро.

String **filePath**; Это поле для пути к упомянутому выше excel файлу с входной информацией для расчета, о котором напишу более подробно. Для каждого поставщика вводится отдельно.

@OneToMany(cascade = {CascadeType.***PERSIST***, CascadeType.***DETACH***,CascadeType.***MERGE***, CascadeType.***REFRESH***}  
 ,mappedBy=**"supplier"**)  
 List<Item> **items**; Об этом поле я уже тоже упоминал. Тут мы храним список товаров для каждого поставщика.

Теперь перейдем к описанию следующего класса – Item – шаблона для описания каждой единицы товара.

**Класс Item**

@Id  
@GeneratedValue(strategy = GenerationType.***IDENTITY***)  
@Column (name = **"id"**)  
**private long id**; - id товара

@Column (name = **"name"**)  
**private** String **name**; - наименование товара из торговой базы

@Column (name = **"article"**)  
**private** String **article**; - артикул товара из торговой базы

@ManyToOne(cascade = {CascadeType.***PERSIST***, CascadeType.***DETACH***,CascadeType.***MERGE***,  
 CascadeType.***REFRESH***})  
@JoinColumn(name = **"supplier\_id"**)

**private** Supplier **supplier**; - поставщик товара. Теоретически можно было бы без него обойтись и не создавать bi-directional связь между объектами классов. Но решил, что так будет удобнее для дальнейшей обработки и вывода данных.

@ElementCollection  
**private** List<Double> **salesPerWeek** = **new** ArrayList<>(52); - упомянутый выше *понедельный* список количества проданных единиц товара за статистический период.

@OneToOne(cascade = {CascadeType.***PERSIST***, CascadeType.***DETACH***,CascadeType.***MERGE***,  
 CascadeType.***REFRESH***})  
@JoinColumn(name = **"price\_id"**)  
**private** Price **price**; - поле цен товара. На данном этапе разработки программы нас интересуют только закупочные цены поставщика, но в дальнейшем можно будет планировать и доходную часть от продаж с помощью отпускных - розничных и оптовых цен.

@OneToOne(cascade = {CascadeType.***PERSIST***, CascadeType.***DETACH***,CascadeType.***MERGE***,  
 CascadeType.***REFRESH***})  
@JoinColumn(name = **"remains\_id"**)  
 **private** Remains **remains**; - необходимое поле для расчета и получения списка прогнозируемого понедельного количества единиц товара на нашем складе. В свою очередь содержит поле **private double currentQty**, куда мы заносим текущие остатки товара, и поле **private** List<Double> **remainsPerWeek,** куда мы заносим расчётные прогнозируемые остатки товара на каждую неделю расчётного периода.

В дальнейшем класс Item можно также дополнить полями, где бы содержалась информация о весе, объеме груза и таможенных пошлинах, как для более точной оценки себестоимости товара, так и для помощи в работе по предоставления необходимой информации таможенному брокеру.

**Класс Parameters**

В него я вынес общие константы, необходимые для расчета

**private static int** *weeksNum* = 52; - количество недель, на которое мы расчитываем бюджет – по умолчанию принято 52, то есть год

**private static double** *expectedSalesGrowthRatio* = 1.0; - коэффициент ожидаемого увеличение/уменьшения продаж в планируемом (расчетном) периоде. Например, наш коммерческий директор ставил в план ежегодное увеличение продаж на 20%, в этому случае *expectedSalesGrowthRatio* = 1.2  
 **private static double** *serviceRatio* = 1.2; - так называемый коэффициент сервиса, на который нужно умножить расчетное (по статистике предыдущего периода) количество товара, чтобы иметь его на остатках на складе с определенной долей вероятности. Вытекает из теории вероятности. Обычно рекомендуется принять 1,2, это позволяет не слишком перегружать бюджет и при этом иметь товар в наличии до следующей поставки с вероятностью 95%. (Вероятность 100% наличия товара не гарантирует ни один коэффициент, так-как всегда есть отличная от нуля вероятность, что какому-то клиенту понадобится срочно столько, что обычно продается за несколько лет.) Для более тонкой настройки расчетов его можно будет вынести в поле поставщика или даже в поле товара, потому что наличие каких-то товаров для нас более критично, и коэффициент сервиса можно принять за 1,5 или даже 2.

Мы рассмотрели основные сущности нашей программы, теперь можно перейти к объяснению логики расчетов – классов из папки **calculation**.

В этой папке находится четыре класса Presenter, Order, ItemOrder и OrderDAO

**Класс Presenter**

Этот класс предназначен для управления расчетом заказов и их сохранением в базу данных.

Поля этого класса – необходимые репозитории для получения и сохранения объектов (**supplierRepository**;**itemRepository**;**orderRepository**;)

А также List<Supplier> **suppliers –** список поставщиков, по которым считаем заказы и бюджет на закупки,

и List<Order> **orders –** сюда мы складываем информацию о рассчитанных заказах для каждого поставщика из списка **suppliers** .

Расчет начинается в методе **public void ordersCalculation()** в котором для каждого поставщика поочередно вызывается метод **order.orderCalculation()** класса **Order**, и результат его действия добавляется в список заказов **orders.**

**Класс Order**

Каждый объект класса Order содержит всю информацию ***о всех*** заказах поставщика. Помимо поля **private** Supplier **supplier**

содержит поля :

**- private** List<ItemOrder> **itemOrders** коллекция для временного хранения объектов Itemorder, в которых находится информация о товаре , его количестве в заказе , и о номере недели, когда этот товар нужно заказать (эта неделя, разумеется, будет одинаковой для всех элементов списка). То есть по сути дела, каждый такой List-список представляет собой сам заказ -– перечень товаров, их количеств и недели для предполагаемого размещения заказа.

- **private** List<List<ItemOrder>> **orders -** коллекция для хранения этих заказов (получился список списков). Этот список содержит полный получившийся перечень заказов поставщика за планируемый (расчетный) период.

Вернемся к расчетам. Итак, в методе **order.orderCalculation()** мы поочередно по каждой неделе расчетного периода проверяем, нужно ли делать на этой неделе заказ поставщику.

Если сумма получившегося заказа(в валюте) меньше **minOrderAmount,** то рассчитанный заказ не сохраняем в список и переходим к следующей неделе. Если больше или равен, то переходим в специальный метод **public void** prepareOrder() , где добавляем заказ в список заказов **orders (orders**.add(**itemOrders**)) и пересчитываем ожидаемые остатки товара (поле Remains товара Item) по неделям после получения только что добавленного заказа (т.е вызываем метод fillItemRemains объектов класса ItemOrder) :

***for*** *(ItemOrder itemOrder :* ***itemOrders****){  
 itemOrder.fillItemRemains(itemOrder.getWeekOfOrderReceiving());  
}*

*//(getWeekOfOrderReceiving() – предполагаемая неделя получения заказа на //наш склад.)*

Рассмотрим весь код метода **orderCalculation()**

**public void** orderCalculation() {

List<Item> items = **supplier**.getItems(); - получаем список товаров поставщика  
 **for** (**int** i = 0; i < *getWeeksNum*(); i++) { итерируемся по каждой неделе расчетного периода  
 **double** sum = 0; - в эту переменную будет считать получившуюся сумму заказа в валюте   
 **itemOrders** = **new** ArrayList<>(); - инициализируем список, каждый элемент будет содержать информацию о том, когда, сколько и какого нужно товара заказывать  
 **for** (Item item : items) { - для каждого товара поставщика  
 ItemOrder itemOrder = **new** ItemOrder(item, i); инициализируем объект  
 ***itemOrder.itemQuantityCalc(); - вызываем метод, в котором содержится логика расчета необходимого количества текущего товара item***  
 **itemOrders**.add(itemOrder); добавляем получившийся заказ товара в общий список заказа по товарам  
 sum += itemOrder.getQuantity() \* item.getPrice().getPurchasePrice(); вычисляем сумму получившегося заказа товара и добавляем к общей сумме заказа  
 }

Пройдясь по всем товарам поставщика и сосчитав всю сумму заказа, сравниваем ее с minOrderAmount (минимальной суммой заказа поставщика)  
 **if** (sum >= **supplier**.getMinOrderAmount()) prepareOrder();

- если эта сумма больше или равна, то подтверждаем (сохраняем) заказ и пересчитываем остатки товаров после его получения на склад.  
 }  
 }  
}

**Класс ItemOrder**

Если объекты класса Order предназначены для подсчета и хранения всех заказов поставщика, то объекты класса ItemOrder предназначены для расчета заказов по каждому конкретному товару.

Содержит поля **private** Item **item**; - поле товара  
**private int week**; - неделя, на которую рассчитывается заказ товара  
  
**private int weekOfOrderReceiving**; - неделя предполагаемого получения товара в случае подтверждения заказа. Рассчитывается в конструкторе по формуле **weekOfOrderReceiving** = week + item.getSupplier().getLeadTime() + item.getSupplier().getDeliveryTime(); где leadTime – срок производства заказа в неделях, deliveryTime – срок транспортировки товара от склада поставщика до нашего склада, в неделях

**private double quantity**; - рассчитываемое количество товара, которое нужно заказать.

К методам объектов этого класса мы обращаемся из методов класса Order.

Класс Itemorder содержит два основных метода - метод ***itemQuantityCalc*** для расчета необходимого количества товара. и метод ***fillItemRemains*** для понедельного перерасчета остатков товара после планируемого поступления заказа на склад.

Опишем код и логику методов

**public void itemQuantityCalc()** { *- к этому методу мы обращаемся из метода ordersCalculation() в цикле итерации по товарам поставщика.*  
**int** weekOfOrderProvision = **weekOfOrderReceiving** + **item**.getSupplier().getOrderPeriod(); - *рассчитываем weekOfOrderProvision - номер недели, до которой нам нужно обеспечить склад данным товаром: (orderPeriod – период между заказами для данного поставщика*)

**double** salesOrderProvision = 0; *в эту переменную будем записывать ожидаемое количество товара, которое предположительно будет продано за период между поступлениями заказов на склад.*

**for** (**int** i = **weekOfOrderReceiving**; i < weekOfOrderProvision; i++) { *(итерируемся по этим неделям)*  
 salesOrderProvision += **item**.getSalesPerWeek().get(i); - *считываем количество проданного товара за i-тую неделю и добавляем к общему количеству*  
 }*выход из цикла*  
 salesOrderProvision \*=*getServiceRatio*()\**getExpectedSalesGrowthRatio*(); - *получаем окончательное количество товара, которое планируем продать за период между поступлениями заказов на склад путем умножения получившейся суммы проданного товара на коэффициент сервиса и коэффициент ожидаемого роста/снижения продаж.*  
 **quantity** = salesOrderProvision - expectedRemainsCalc(); - *вычисляем необходимое количество товара для заказа (метод* expectedRemainsCalc()

*возвращает ожидаемые остатки товара на неделю поступления заказа)*

*}*

**if** (**quantity**<0) **quantity** = 0; -*теоретически может получиться отрицательное количество, если возвраты от покупателей на склад были больше продаж, отрицательное количество мы заказать не можем, ставим 0.*}

Рассмотрим код другого важного метода класса - *fillItemRemains* . К нему мы обращаемся из метода *prepareOrder()* класса *Order* при подтверждении рассчитанного заказа, так как после поступления заказ на склад остатки необходимо пересчитать.

**public void fillItemRemains**(**int** weekNum) { - *weekNum - номер недели запланированного получения заказа*  
 Remains remains = **item**.getRemains(); *- получаем у товара item поле остатков Remains* List <Double> remainsPerWeek = remains.getRemainsPerWeek(); - получаем список расчётных понедельных остатков товара  
 List <Double> sales = **item**.getSalesPerWeek(); - получаем список понедельных продаж товара из статистического периода.  
  
 **double** remain = remainsPerWeek.get(weekNum); - *получаем планируемый остаток товара на неделю запланированного размещения заказа* remainsPerWeek.set(weekNum, remain + **quantity**); *пересчитываем его (увеличиваем на количество заказанного товара) и сетим в коллекцию его новое значение.*  
  
 **for** (**int** i = weekNum+1; i<remainsPerWeek.size(); i++){ *проходим со следующей за получением заказа недели и до конца расчетного периода* **double** salesThisWeek = sales.get(i)\* *getServiceRatio*()\**getExpectedSalesGrowthRatio*(); - *пересчитываем ожидаемые продажи на i-той неделе с учетом коэффициента сервиса и коэффициента ожидаемого изменения спроса.* **double** remainsThisWeek = remainsPerWeek.get(i-1) - salesThisWeek; *пересчитываем остаток товара на этой неделе (точнее в конце этой недели) с учетом ожидаемых продаж на этой неделе.*  
 **if** (remainsThisWeek<0) remainsThisWeek = 0; *//остаток товара не может быть отрицательным* remainsPerWeek.set(i,remainsThisWeek); *и сетим его в коллекцию* } *выходим из цикла*  
 remains.setRemainsPerWeek(remainsPerWeek); *и сетим получившуюся пересчитанную коллекцию ожидаемых понедельных остатков в поле remains поля item.* }

Таким образом я постарался подробно объяснить процесс расчета необходимых заказов по каждому поставщику в течение года.

Далее перейдем к описанию пользовательского интерфейса и подготовки необходимых внешних данных для расчетов.

**2.2. Разработка пользовательского интерфейса**

Программа выполнена в виде Spring Boot проекта. Spring Boot мы изучали на семинарах GB, он позволяет наиболее простым и быстрым способом создать web-приложение, требуя от разработчиков минимум усилий по его настройке и написанию кода. Несмотря на то, что мой проект является по сути десктопным, сыграла свою роль простота настройки зависимостей через starter-пакеты и удобство работы с пользовательским интерфейсом через формы и таблицы шаблонизатора **Thymeleaf.**

настройке Maven – pom.xml я указал следующие зависимости

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>  
 *- это основная зависимость spring boot, позволяющая проекту работать как web приложение*

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
</**dependency**>

*- зависимость для работы с репозиториями. Простота работы с базами данных – еще одно достоинство Spring Boot*

<**dependency**>  
 <**groupId**>com.h2database</**groupId**>  
 <**artifactId**>h2</**artifactId**>  
</**dependency**>  
*- зависимость для подключения базы данных.*

<**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**optional**>true</**optional**>

</**dependency**>  
*- зависимость для подключения утилиты Lambok – позволяет сократить код в программе, автоматически создает необходимые конструкторы, геттеры и сеттеры..*

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**>

</**dependency**>  
*- зависимость для подключения шаблонизатора web – страниц.*

В качестве базы данных я выбрал СУБД H2, так как считаю она наилучшим образом подходит для данного проекта, по крайней мере на данном этапе. Она сочетает в себе все удобства наличия базы данных в Spring Boot проекте, и при этом не оставляет после работы приложения физической информации на жестком диске или ином носителе, так-как целиком хранится в оперативной памяти компьютера. Это важно, потому что информация, с которой производим расчеты, как правило является коммерческой и конфиденциальной, и сохранение лишних данных за пределами основной торговой базы не желательно. А данные, которые мы получим в результате работы программы, мы можем легко скопировать из браузера и сохранить в любой удобный офисный word или excel формат в отдельный файл или файлы.

<**dependency**>  
 <**groupId**>org.apache.poi</**groupId**>  
 <**artifactId**>poi</**artifactId**>  
 <**version**>5.0.0</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.apache.poi</**groupId**>  
 <**artifactId**>poi-ooxml</**artifactId**>  
 <**version**>5.0.0</**version**>  
</**dependency**>

*- необходимые для проекта зависимости, позволяющие загружать в работу данные из эксель – таблиц*

После подключения Maven зависимостей, создаем Spring Boot – проект

@SpringBootApplication  
**public class** VedmasterApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(VedmasterApplication.**class**, args);  
 }  
}

Аннотация @SpringBootApplication помечает класс, который запускает Spring Boot.

Осталось наполнить проект контроллерами, сервисами, репозиториями, создать html страницы с формами для ввода данных и таблицами для вывода результатов, и интегрировать в проект рассмотренные выше классы для расчетов.

В проекте я подготовил следующие контроллеры

SupplierController – для введения и вывода данных по поставщикам

ItemController – для вывода списка товаров

CalculationController – для запуска расчетов

OrderController – для вывода рассчитанных заказов и прочих результатов расчетов

**SupplierController**

Содержит следующие методы :

- Метод setSupplierDetails – с него следует начинать пользование программой, вызывается в браузере по адресу <http://localhost:8080/supplier>

В результате его действия вызывается страница для заполнения деталей поставщика - **supplier-set.html**

@GetMapping(**"/supplier"**)

String setSupplierDetails(Model model)

{  
 model.addAttribute(**"supplier"**, **new** SupplierRequest());  
 **return "supplier-set"**;  
 }

**-** Метод showSupplierDetails выводит на экран информацию о поставщике (страница **supplier-view.html)**

@GetMapping(**"/showSupplierDetails"**)  
 String showSupplierDetails(@ModelAttribute(**"supplier"**) SupplierRequest supplier) {  
 **supplierService**.addSupplier(supplier);  
 **return "supplier-view"**;  
 }

**-** Метод suppliers выводит на экран список поставщиков (страница **suppliers.html)**   
 @GetMapping(**"/suppliers"**)  
 **public** String suppliers(Model model) {  
 List<Supplier> suppliers = **supplierService**.getSuppliers();  
model.addAttribute(**"suppliers"**, suppliers);  
 **return "suppliers"**;  
 }

**-** Метод filePath выводит форму для ввода пути к эксель-файлу с информацией по поставщику (страница **filePath.html**)

@GetMapping(**"/filePath"**)  
 **public** String filePath(Model model) {  
 model.addAttribute(**"config"**, **new** Config());  
 **return "filePath"**;  
 }  
**-** Метод readDetails показывает введенный путь к эксель-файлу с информацией по поставщику (страница **second-view.html**) и запускает метод **configService**.readFile для чтения из этого файла.

@GetMapping(**"/showFilePath"**)  
 String readDetails (@ModelAttribute(**"config"**) Config config) {  
 **try** {  
 **configService**.readFile(config);  
 } **catch** (FileNotFoundException e) {  
 **throw new** RuntimeException(e);  
 }  
 **return "second-view"**;  
 }  
}

**ItemController**

Здесь создан пока единственный метод showItems, выводящий информацию обо всех рассчитываемых товарах всех продавцов (страница items.html.) Нужен, чтобы удостовериться, что все данные правильно перенеслись из ексель-файлов в базу данных.

**public class** ItemController {

@Autowired  
 ItemService **itemService**;  
 @GetMapping(**"/items"**)  
 **public** String showItems(Model model){  
 List<Item> items = **itemService**.getItems();  
 model.addAttribute(**"items"**,items);  
 **return "items"**;  
 }

Планировался еще один метод @GetMapping(**"/items/{id}"**)  
**public** String showItemDetails(@PathVariable **long** id), выводящий информацию о товаре по его id, но пока не вижу его практической пользы для проекта.

**OrderController**

Содержит метод showOrders, возвращает станицу **orders.html ,** где отображается общая информация о всех рассчитанных заказах

@GetMapping(**"/orders"**)  
 **public** String showOrders(Model model){  
 List<OrderDAO> orders = **orderService**.getOrders();  
 model.addAttribute(**"orders"**,orders);  
 **return "orders"**;  
 }  
Метод showItemDetails выводит более подробную информацию о заказе по id с перечнем и количеством товаров в заказе, ценами и суммами. Для удобного отображения информации использованы объекты вспомогательных классов OrderDAO и ItemOrderView.

@GetMapping(**"/orders/{id}"**)  
 **public** String showItemDetails(@PathVariable **long** id, Model model){  
 OrderDAO order = **orderService**.getById(id);  
  
 **long** di = order.getId();  
 String supplierName = order.getSupplierName();  
 List<String> articles = order.getItemArticles();  
 List<String> itemNames = order.getItemNames();  
 List<Double> itemQuantities = order.getItemQuantities();  
 List <Double> itemPurchasePrices = order.getItemPurchasePrices();  
 List<Double> amounts = order.getAmounts();  
 **int** week = order.getWeek();  
 **double** orderTotalAmount = order.getOrderTotalAmount();  
  
 List<ItemOrderView> itemOrderViews = **new** ArrayList<>();  
  
 **for** (**int** i = 0; i<articles.size();i ++){  
  
 ItemOrderView itemOrderView = **new** ItemOrderView();  
 itemOrderView.setId(di);  
 itemOrderView.setSupplierName(supplierName);  
 itemOrderView.setItemArticle(articles.get(i));  
 itemOrderView.setItemName(itemNames.get(i));  
 itemOrderView.setItemQuantity(itemQuantities.get(i));  
 itemOrderView.setItemPurchasePrice(itemPurchasePrices.get(i));  
 itemOrderView.setAmount(amounts.get(i));  
 itemOrderView.setWeek(week);  
 itemOrderView.setOrderTotalAmount(orderTotalAmount);  
  
 itemOrderViews.add(itemOrderView);  
 }  
 model.addAttribute(**"itemOrderViews"**, itemOrderViews);  
 **return "order"**;  
 }  
}

Теперь рассмотрим кратко слой сервисов – классов, обслуживающих контроллеры и содержащие основную логику процессов.

**ConfigService**

Содержит единственный метод ***readFile***. Предназначен для считывания из файла с исходной информацией и передачи ее в supplierService для сохранения в базу данных.

С него, можно сказать, начинается основная работа приложения, после того как мы завели путь к файлу с исходной информацией.

@Service  
@RequiredArgsConstructor  
@Data

**public class** ConfigService {

@Autowired  
 SupplierService **supplierService**;

**public void** readFile(Config config) **throws** FileNotFoundException {  
 FileInputStream file = **new** FileInputStream(config.getFileName());  
 XSSFWorkbook workbook = **null**;  
 **try** {  
 workbook = **new** XSSFWorkbook(file);  
 } **catch** (IOException ex) {  
 **throw new** RuntimeException(ex);  
 }  
 XSSFSheet sheet = workbook.getSheetAt(0);  
 Iterator<Row> rowIterator = sheet.iterator();  
 List<ItemRequest> requestList = **new** ArrayList<>();  
  
 **while** (rowIterator.hasNext()) {  
 Row row = rowIterator.next();  
 ItemRequest itemRequest = **new** ItemRequest();  
 PriceRequest priceRequest = **new** PriceRequest();  
 RemainsRequest remainsRequest = **new** RemainsRequest();  
  
 itemRequest.setName(row.getCell(0).getStringCellValue());  
 itemRequest.setArticle(row.getCell(1).getStringCellValue());  
 priceRequest.setPurchasePrice(row.getCell(2).getNumericCellValue());  
 itemRequest.setPriceRequest(priceRequest);  
  
 remainsRequest.setCurrentQty(row.getCell(3).getNumericCellValue());  
 itemRequest.setRemainsRequest(remainsRequest);  
  
 **for** (**int** i=4; i<=Parameters.*getWeeksNum*()\*2+5; i++) {  
 **if** (row.getCell(i)==**null**) itemRequest.getSalesPerWeek().add(0d);  
 **else** itemRequest.getSalesPerWeek().add(row.getCell(i).getNumericCellValue());  
  
 }  
  
 requestList.add(itemRequest);  
  
 }  
 **supplierService**.addItems(requestList);  
 }  
}

**SupplierService** связан с SupplierController и содержит методы для работы с репозиториями для хранения данных по поставщикам и их товарам

@Service  
@RequiredArgsConstructor  
@Data

**public class** SupplierService {  
 **private final** JpaSupplierRepository **supplierRepository**;  
 **private final** JpaItemRepository **itemRepository**;

- метод addSupplier добавляет поставщика в базу данных. Вызывается из SupplierController и использует объект вспомогательного класса SupplierRequest, в который пользователь заносит данные из формы соотвтетствующей html страницы.

**public void** addSupplier(SupplierRequest request){  
 Supplier supplier = **new** Supplier(request.getName(),request.getMinOrderAmount(),request.getLeadTime(),  
 request.getDeliveryTime(), request.getOrderPeriod(),request.getDeliveryCostFactor());  
 **supplierRepository**.save(supplier);  
 }

Метод getSuppliers – получения списка поставщиков из репозитория, чтобы удостоверится, что все они занесены в базу и информация по ним корректно заведена.  
 **public** List<Supplier> getSuppliers(){  
 **return supplierRepository**.findAll();  
 }

Метод addItems формирует к только что заведенному поставщику список его товаров, а каждому товару «сетит» его поля. Использует в качестве параметра список объектов вспомогательного класса   
 **public void** addItems(List<ItemRequest> requestList){  
 List<Supplier> suppliers = getSuppliers();  
 Supplier supplier = suppliers.getLast();  
 List<Item> items = **new** ArrayList<>();  
 **for** (ItemRequest request:requestList){  
 Item item = **new** Item();  
 Price price = **new** Price();  
price.setItem(item);  
 item.setName(request.getName());  
 item.setArticle(request.getArticle());  
 item.setId(Item.*sequence*++);  
 item.setSupplier(supplier);  
  
 price.setPurchasePrice(request.getPriceRequest().getPurchasePrice());  
 item.setPrice(price);  
  
 item.setSalesPerWeek(request.getSalesPerWeek());  
 Remains remains = **new** Remains(request.getRemainsRequest().getCurrentQty());  
 remains.setItem(item);  
 **for** (**int** i = 1; i< Parameters.*getWeeksNum*(); i++) {  
 **double** remain = remains.getRemainsPerWeek().get(i - 1) - item.getSalesPerWeek().get(i);  
 **if** (remain < 0) remain = 0;  
 remains.getRemainsPerWeek().set(i, remain);  
 }  
 item.setRemains(remains);  
 items.add(item);  
 }  
 supplier.setItems(items);  
 **supplierRepository**.save(supplier);  
 }  
}

Мы рассмотрели основной код , перейдем к описанию пользования программы.

**3. Инструкция по работе с программой.**

Перед началом работы необходимо подготовить эксель – файлы с данными для каждого поставщика. Данные, например, можно легко выгрузить из отчетов 1С. Эксель-файл из себя представляет таблицу с данными по товарам поставщика. Нужно обратить внимание на порядок столбцов таблицы

-1 столбец – наименование товара

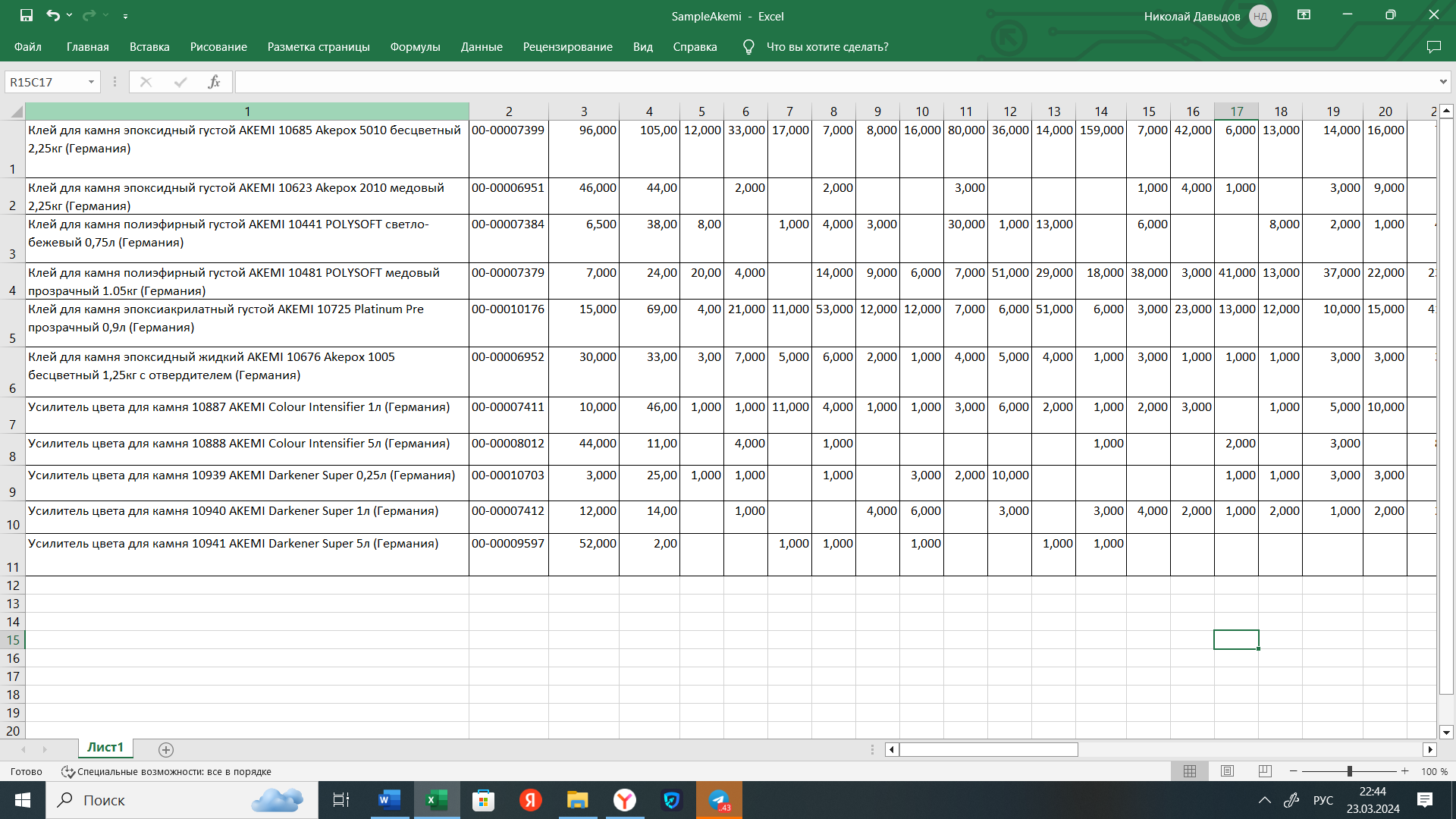
-2 столбец – артикул

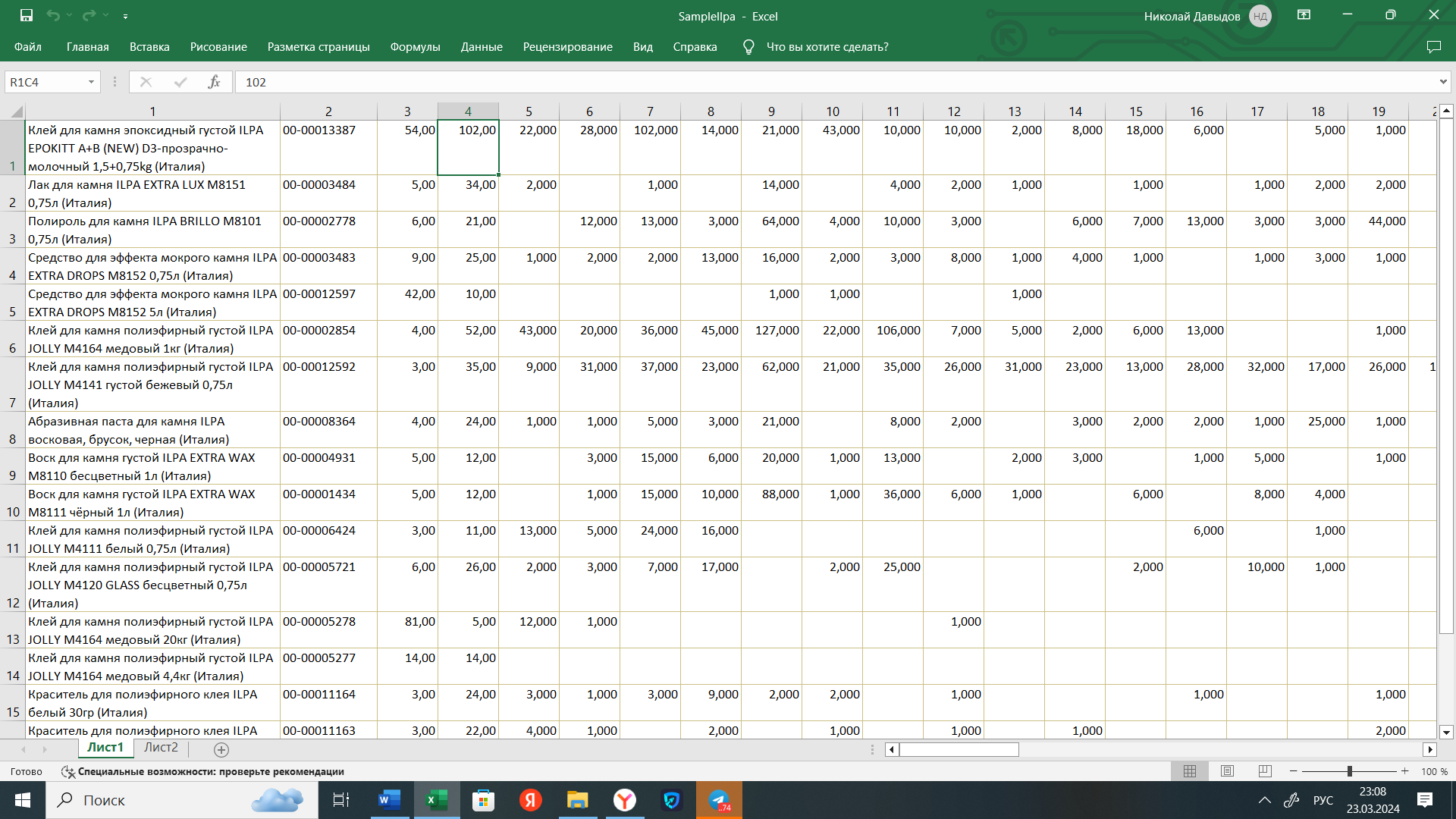
-3 столбец – закупочная цена товара

-4 столбец – текущие остатки товара

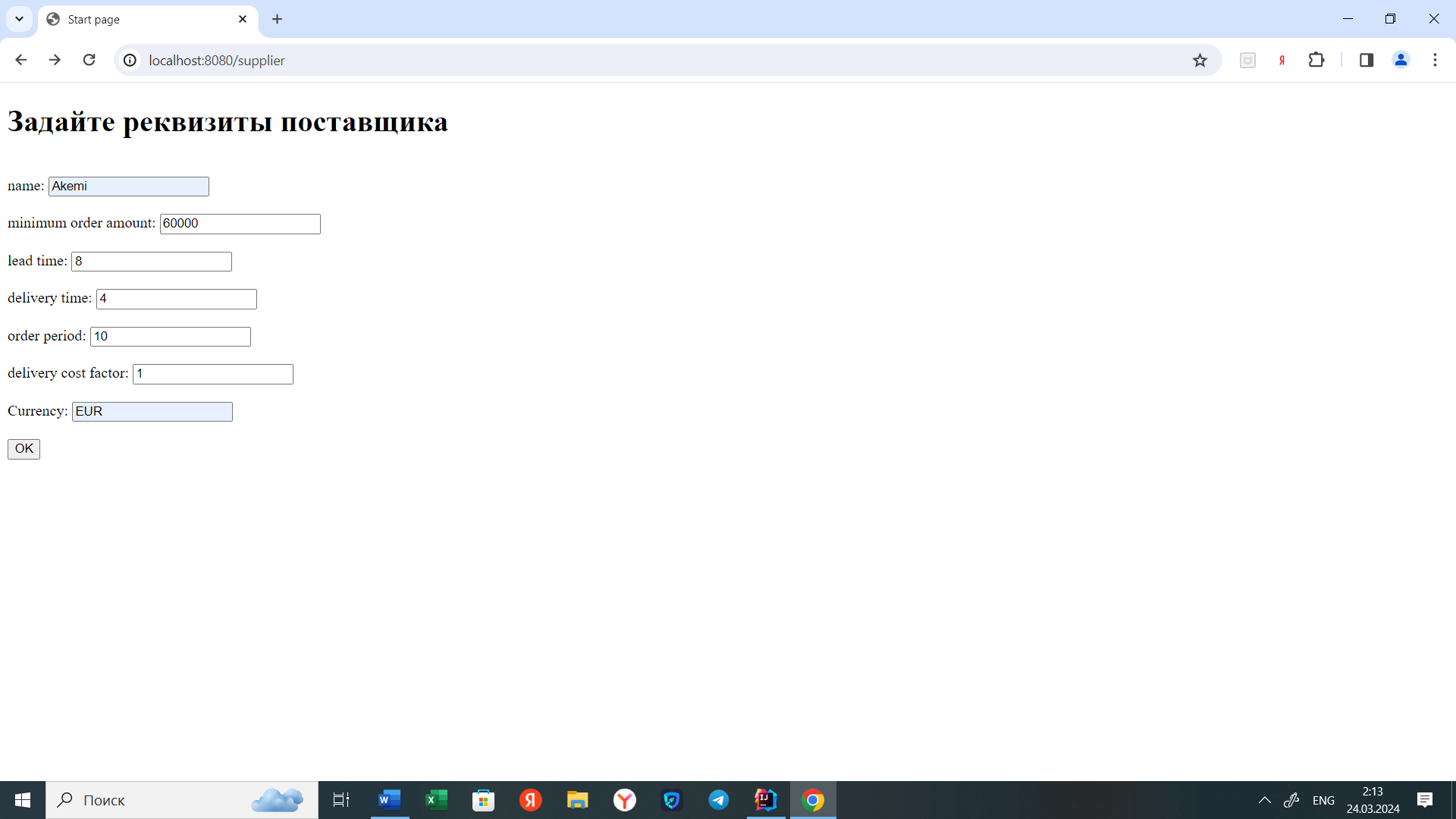
Следующие столбцы показывают количество проданного товара за каждую неделю в предыдущем периоде, взятом для статистики. Количество недель должно быть больше расчетного периода как минимум на величину **orderPeriod,** для расчета необходимости заказа в конце расчетного периода.

Пример файла – для поставщика Akemi – наименование товара, его артикул, цена закупки в Евро, текущие остатки на складе, и продажи в штуках по неделям статистического периода

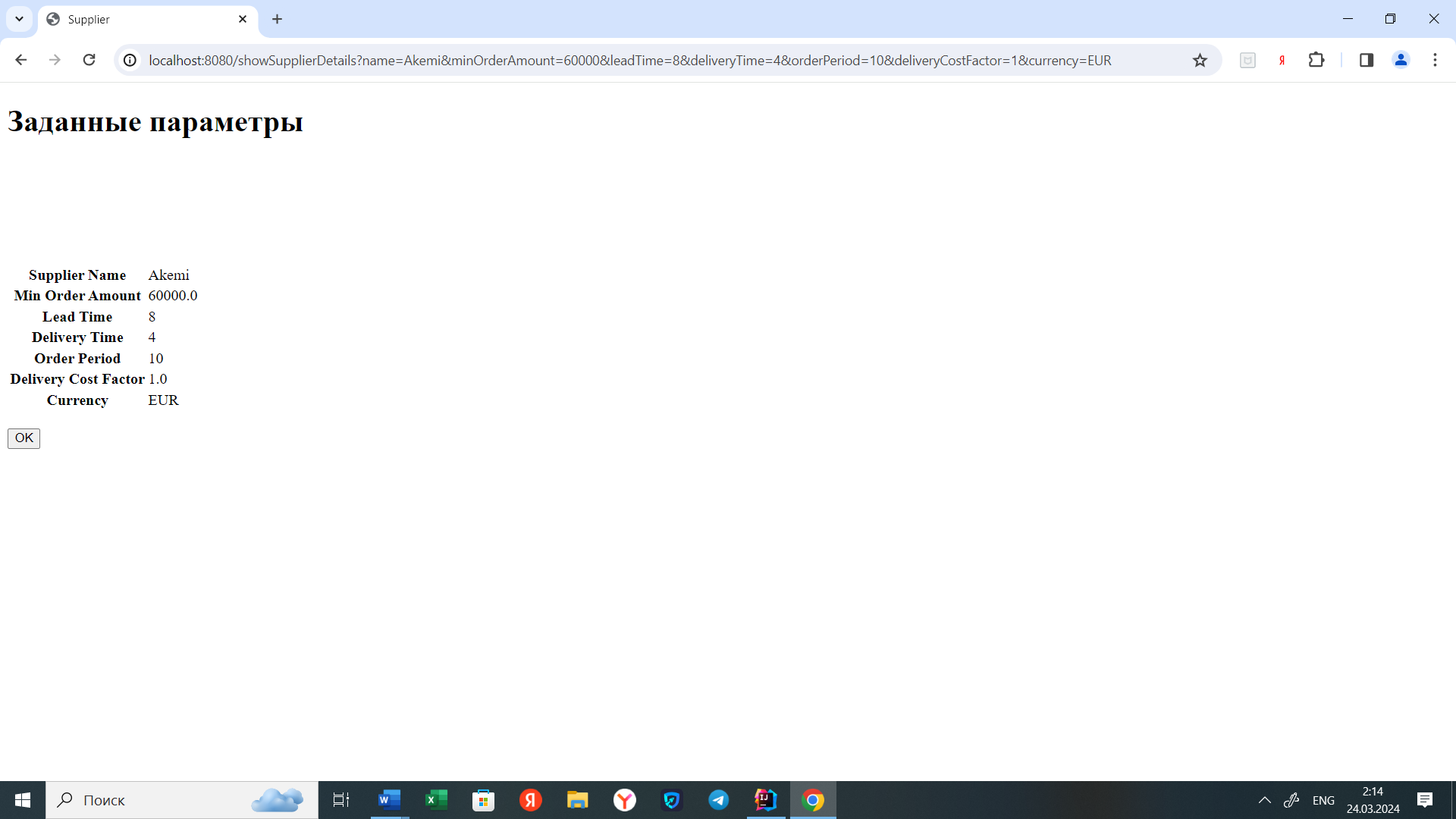


Аналогичный файл для другого поставщика (компания ILPA)

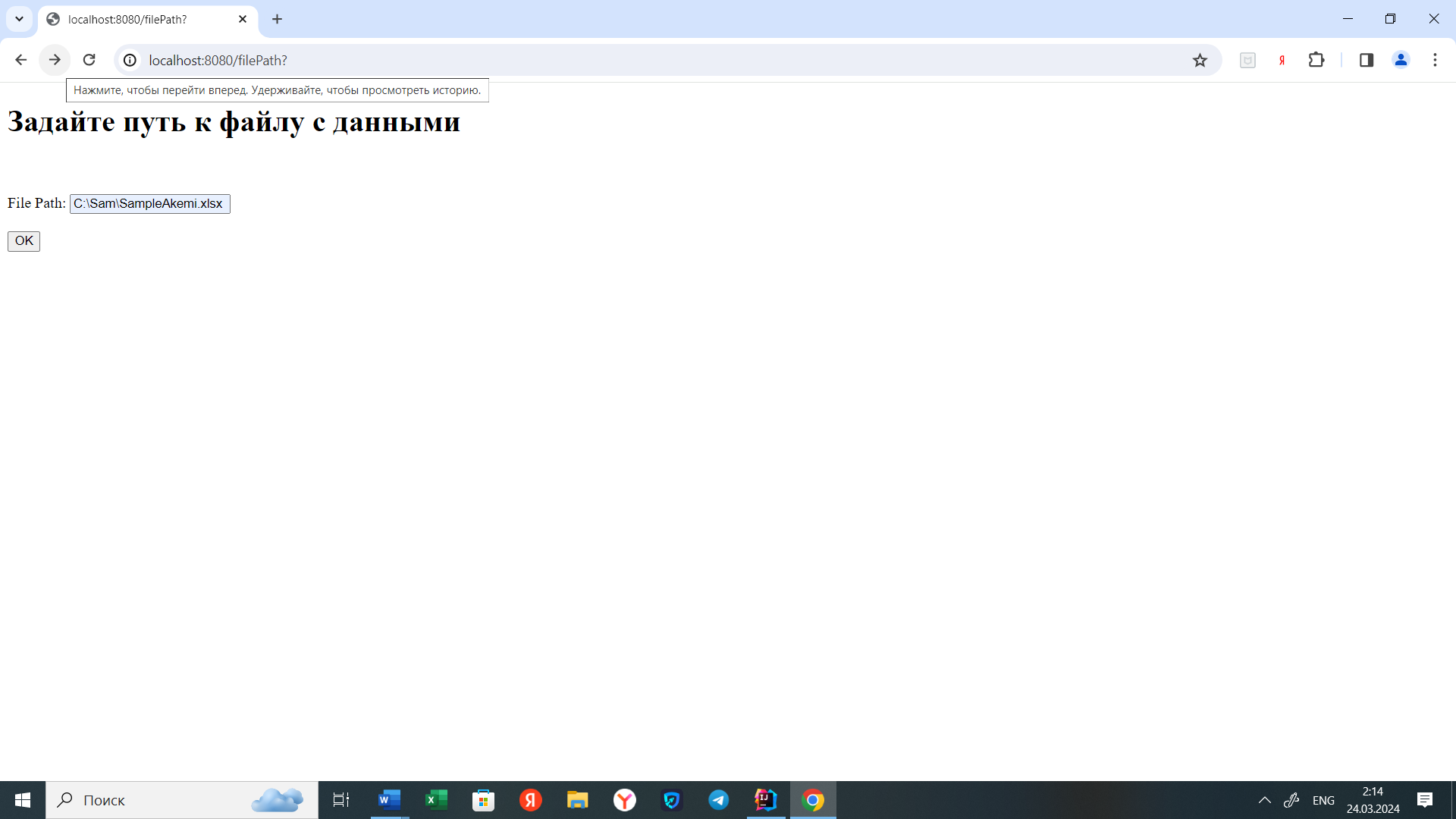
После подготовки файлов запускаем программу набираем в адресной строки браузера [***http://localhost:8080/supplier***](http://localhost:8080/supplier) и заносим данные по поставщику AKEMI в соответствующие поля открывшейся формы :



нажимаем кнопку ОК.

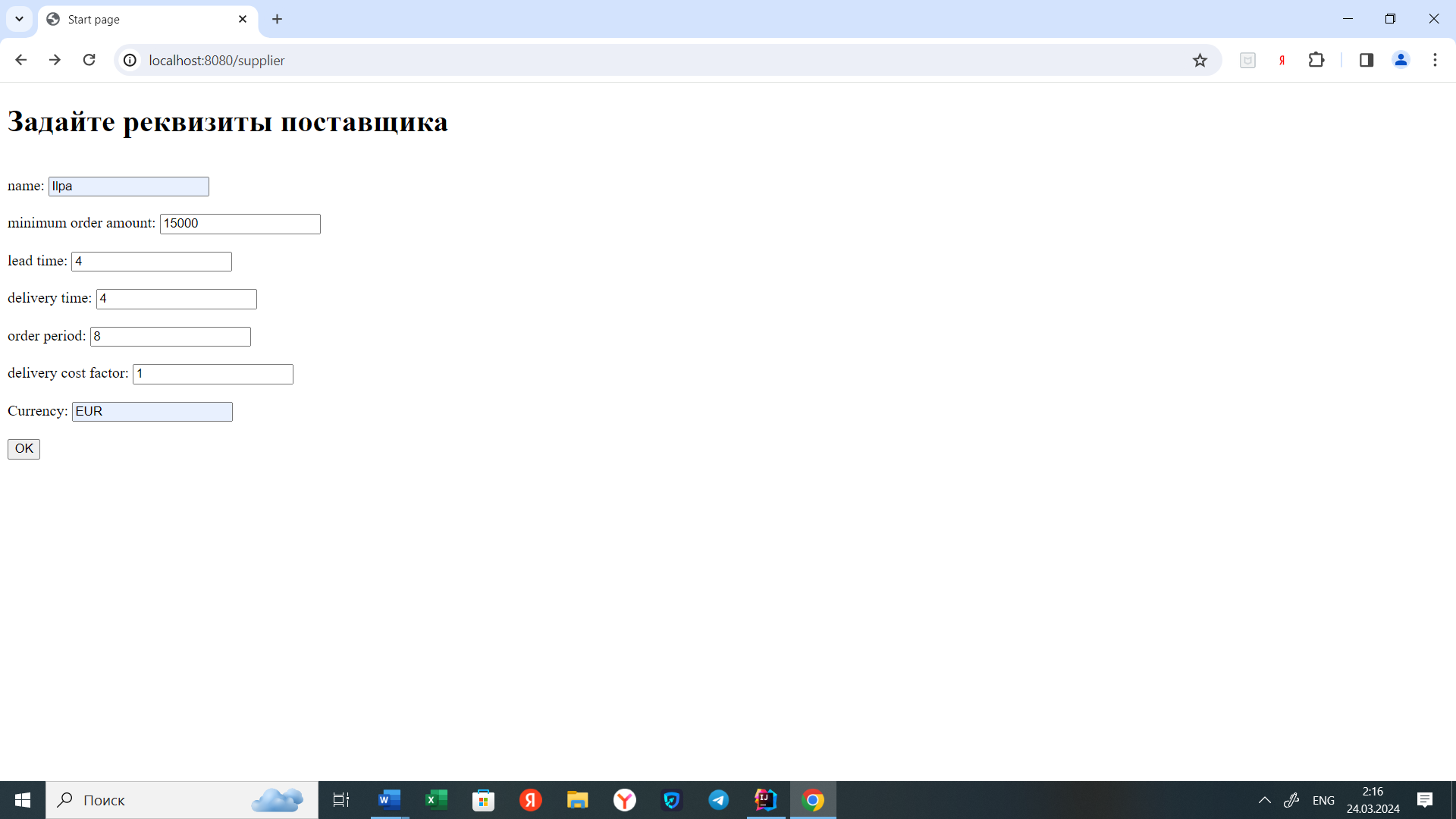


Нажимаем ОК еще раз - открывается поле для ввода пути к подготовленному файлу с данными:

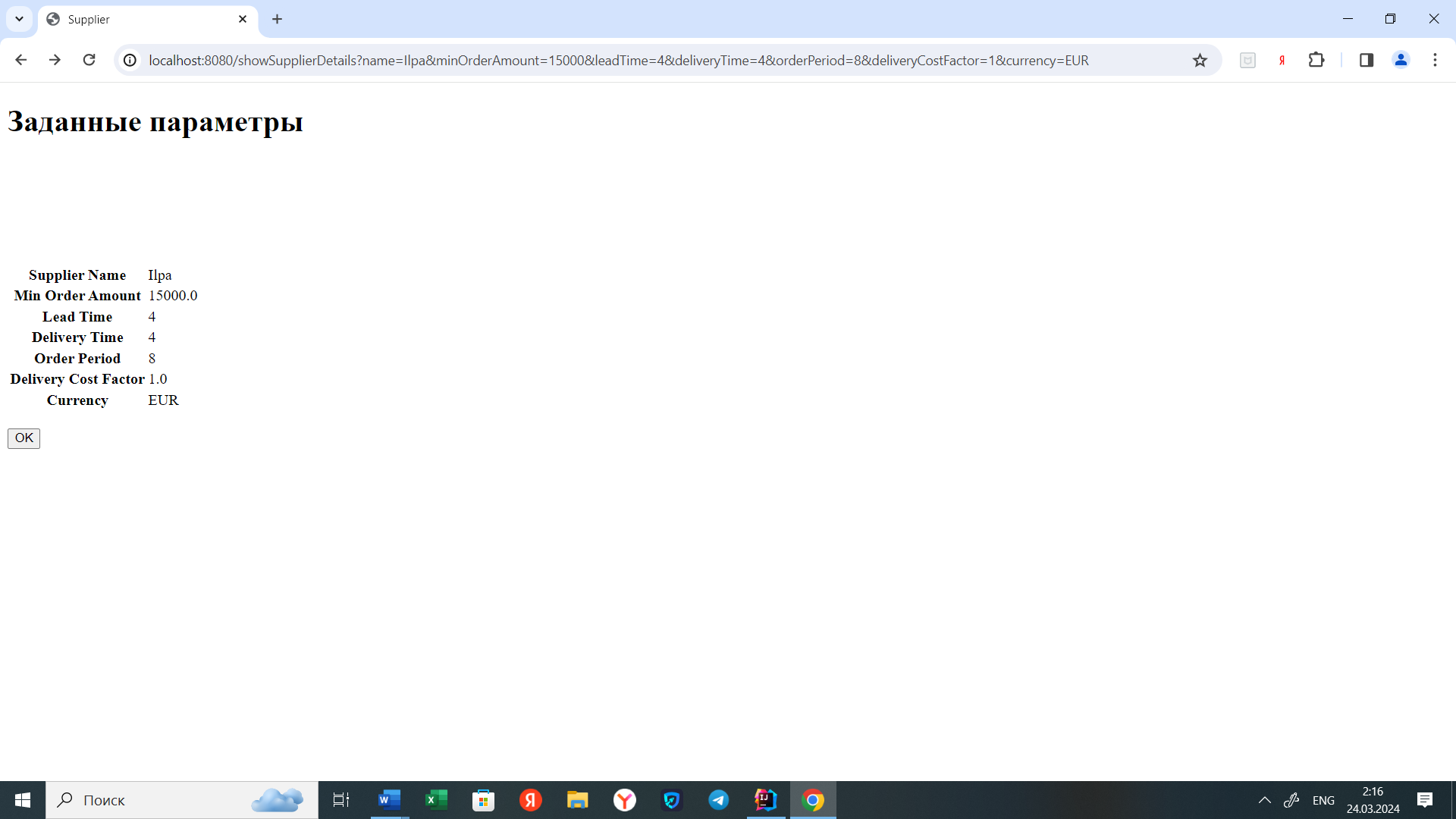


Вводим путь и нажимаем ОК

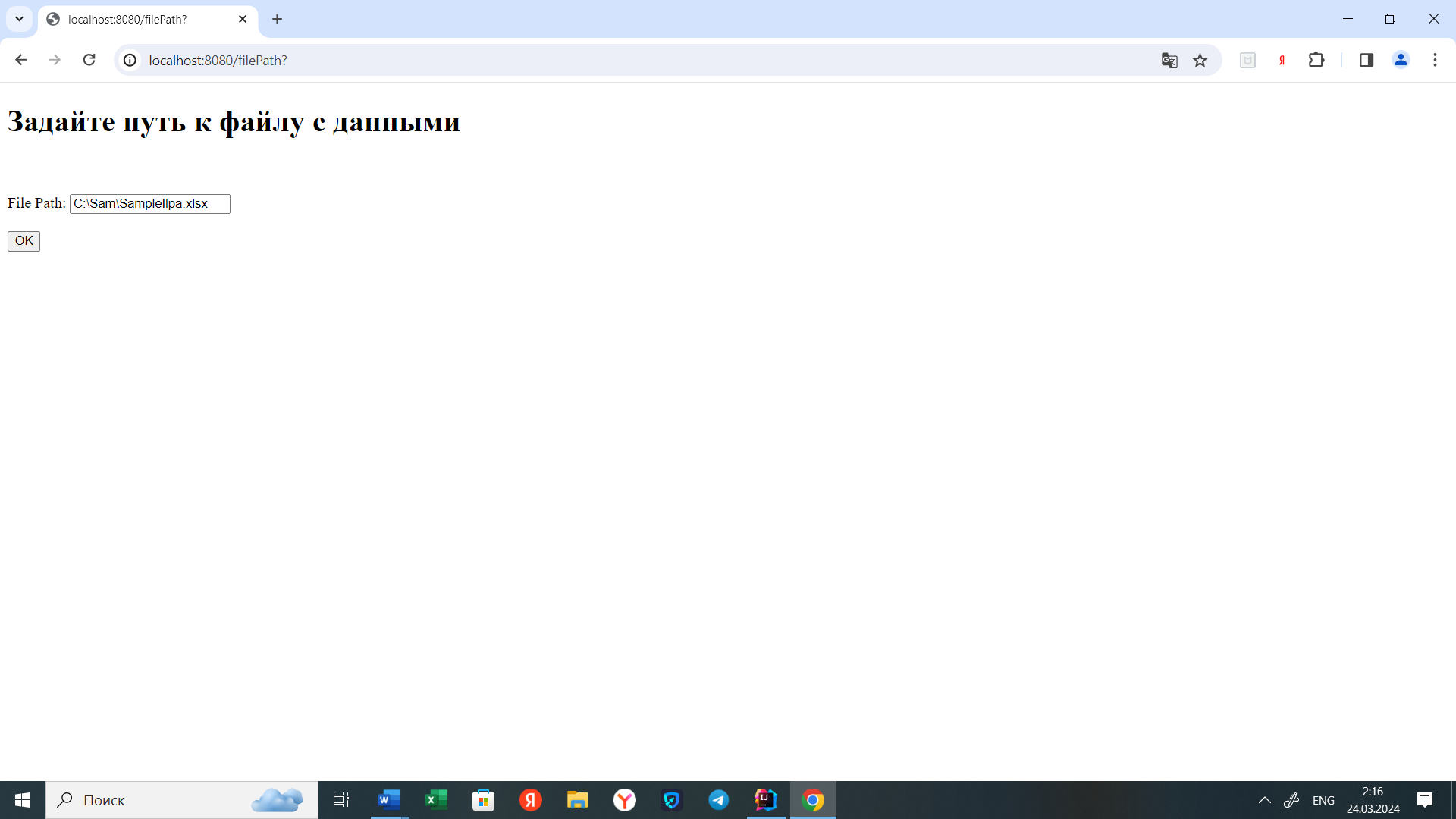
Если хотим рассчитать закупки по еще одному поставщику возвращаемся на адрес [***http://localhost:8080/supplier***](http://localhost:8080/supplier)и также заводим все данные по поставщику ILPA



Нажимаем OK



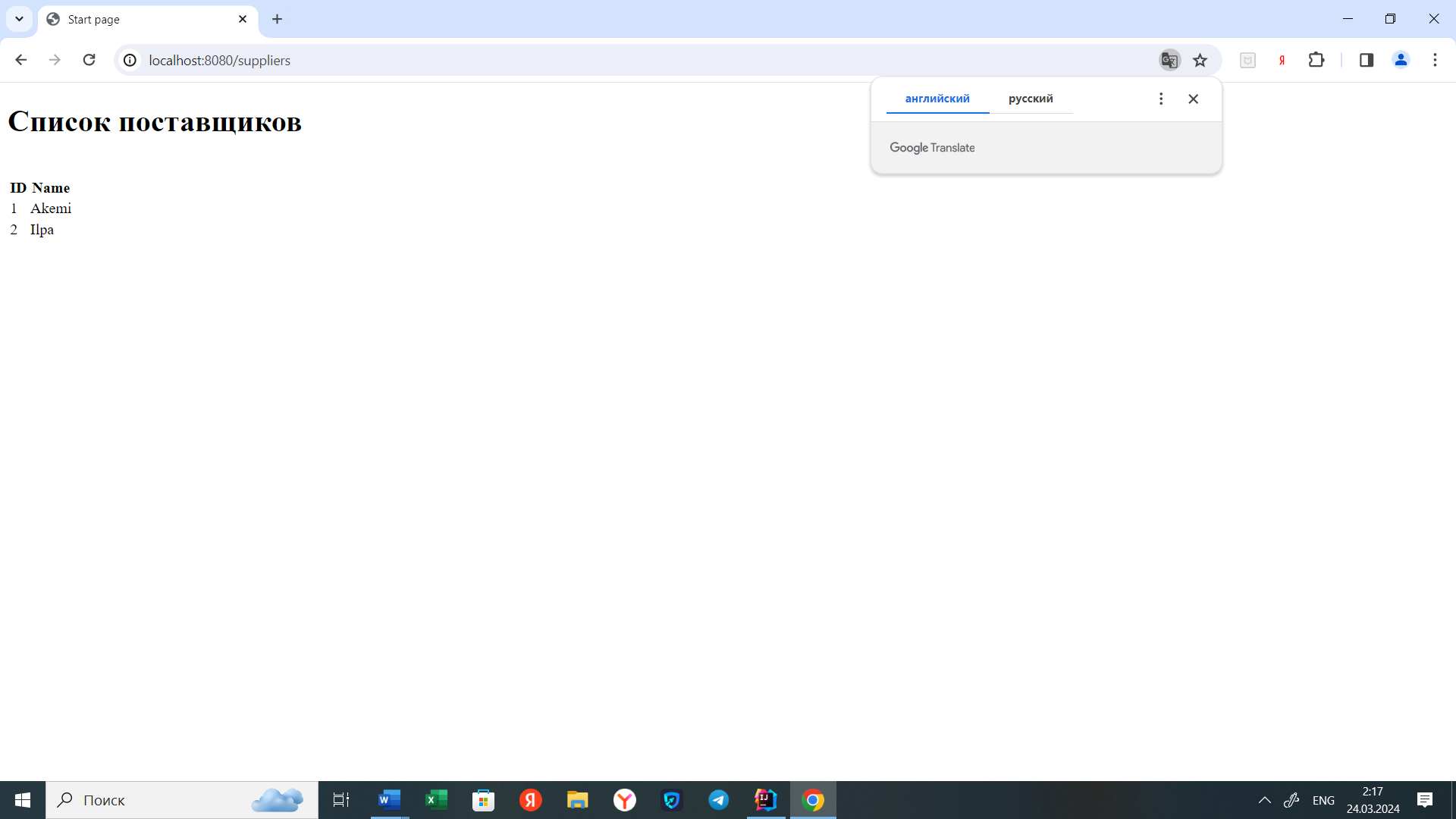
Еще раз ОК и вводим путь к файлу. Нажимаем OK.



Если есть еще поставщики, то необходимо повторить эти шаги для их всех.

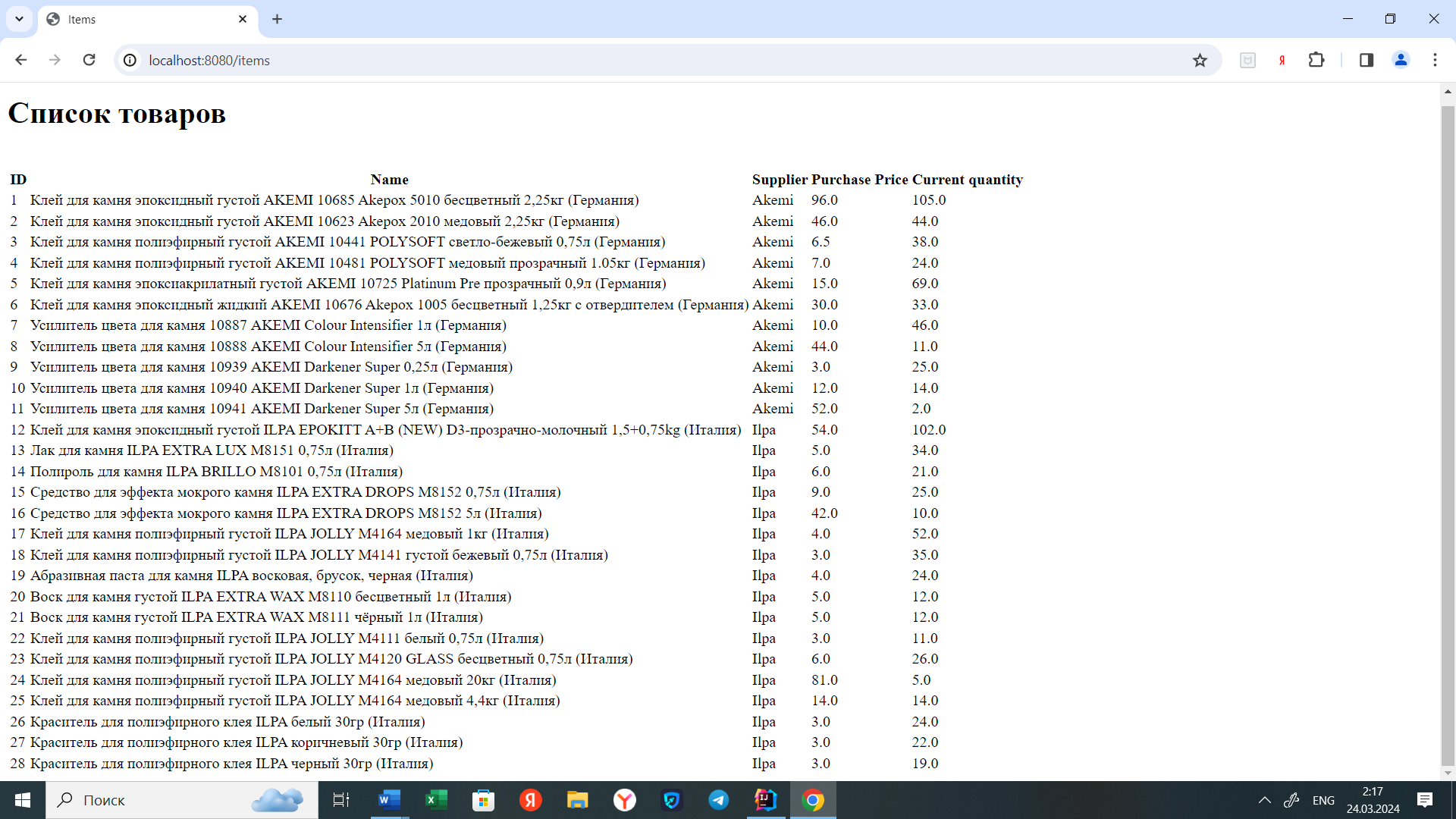
Когда все поставщики заведены, убедимся, что все данные корректно считались из файлов

Набираем в адресной строке браузера <http://localhost:8080/suppliers> - получаем список поставщиков



Набираем в адресной строке браузера <http://localhost:8080/items>

- получаем список товаров



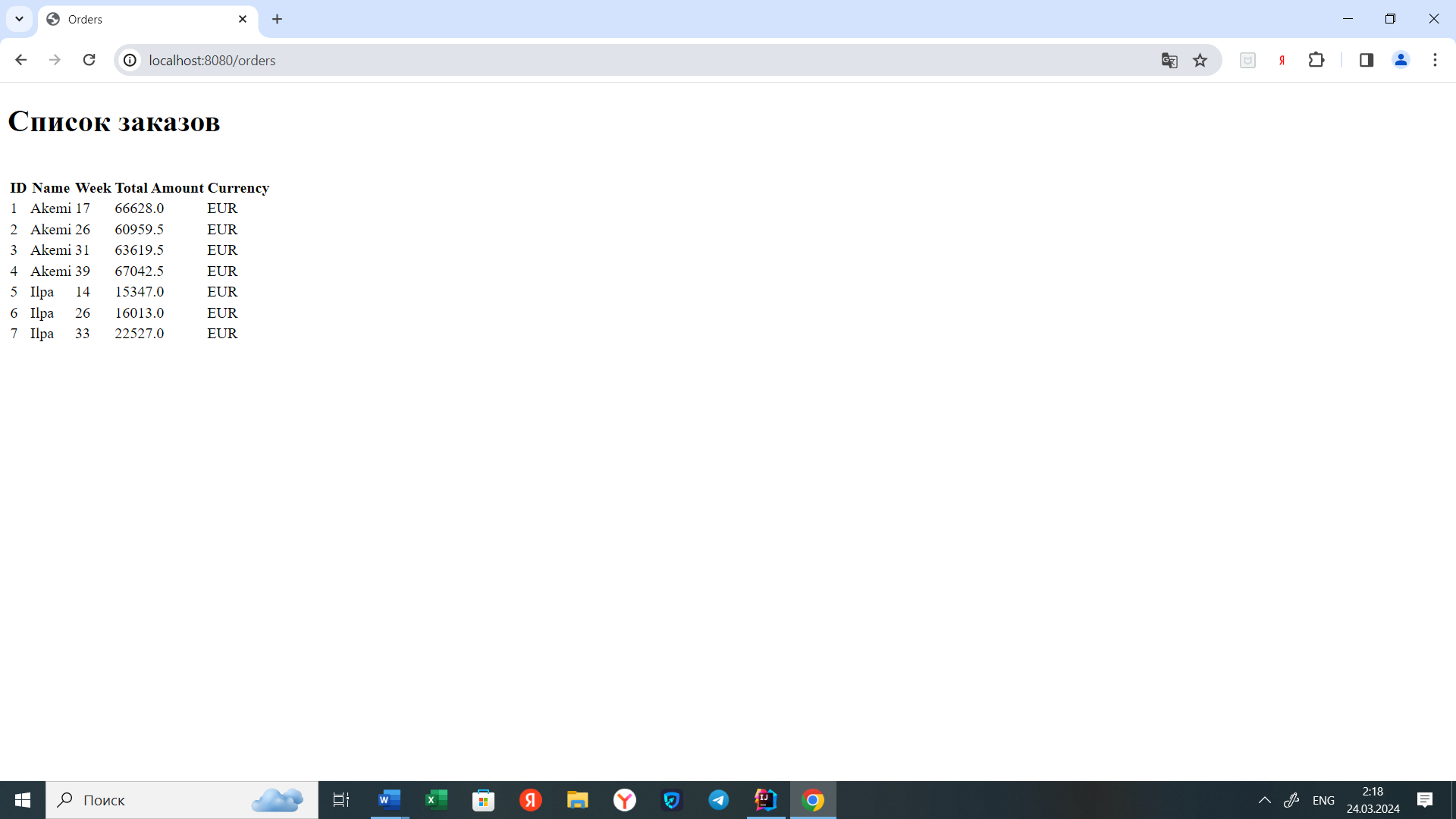
Теперь все готово для расчетов заказов.

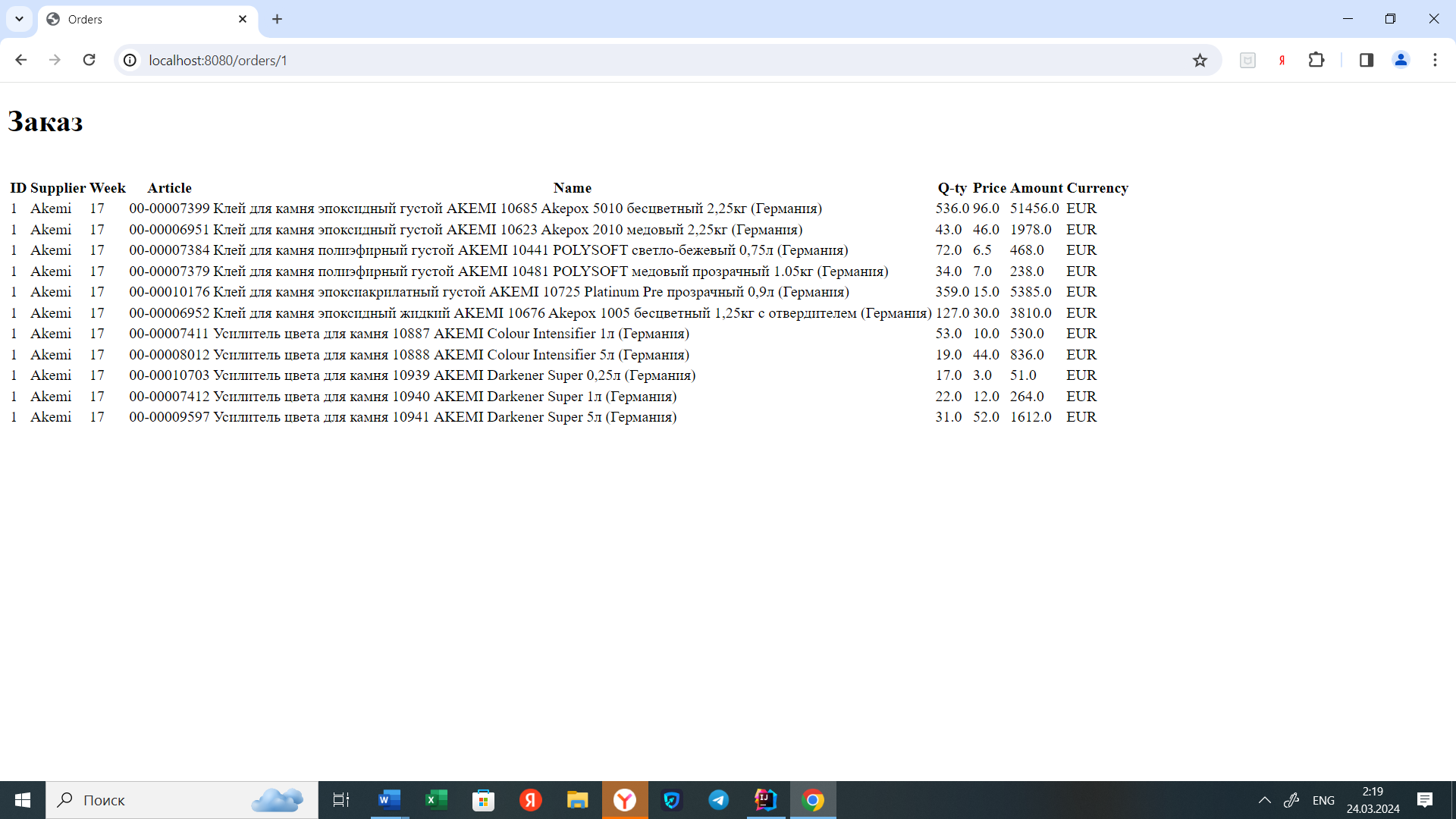
Набираем в адресной строке <http://localhost:8080/calc>

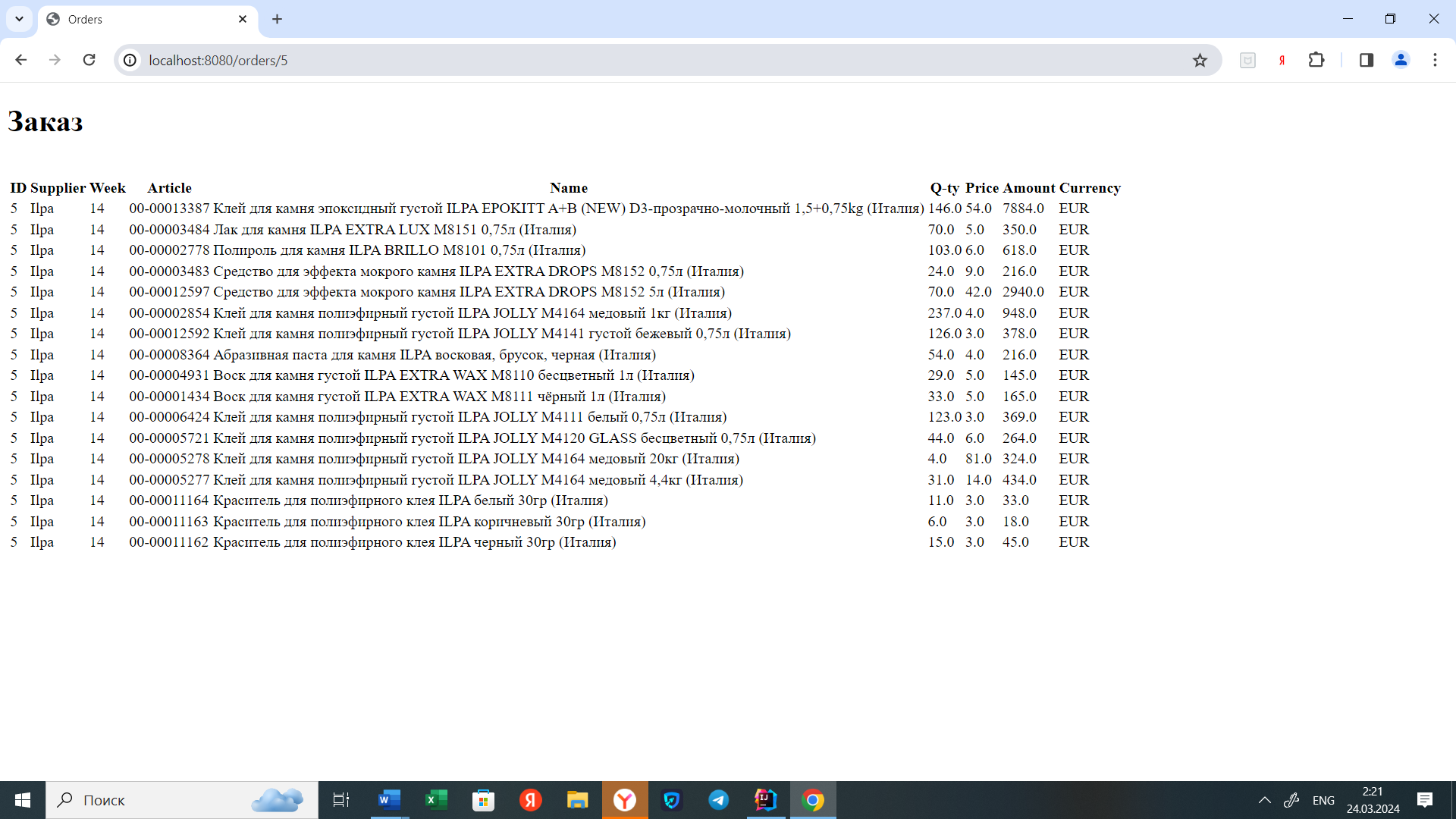
Чтобы посмотреть получившиеся заказы

набираем в адресной строке <http://localhost:8080/orders>

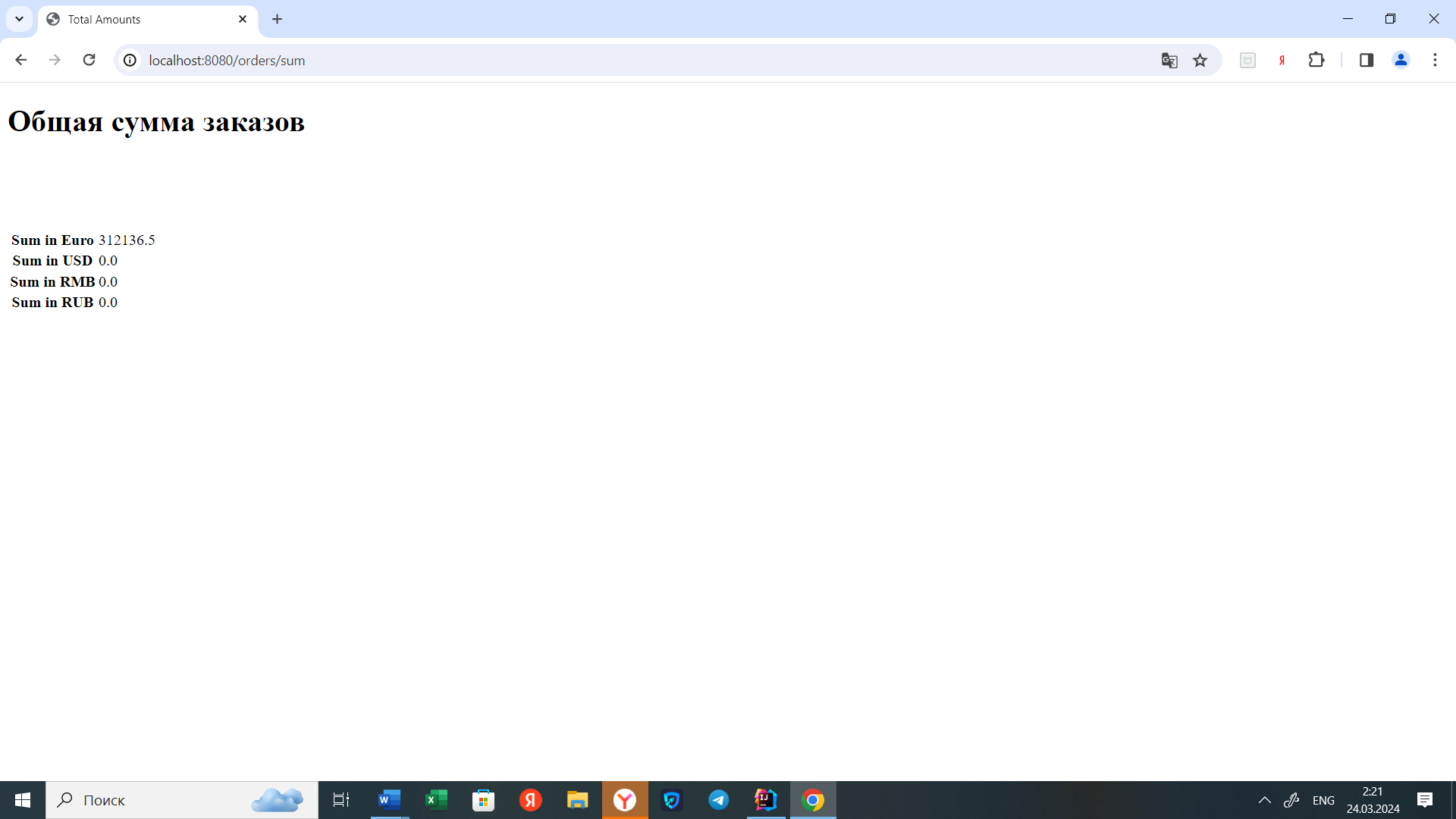
Получаем информацию о неделе, на которой надо будет сделать заказ, какому поставщику и на какую сумму в валюте.



Деталировку заказов можно посмотреть по адресу <http://localhost:8080/orders/1> (где последняя цифра - id заказа из таблицы <http://localhost:8080/orders>)

Выводится информация о поставщике, номере недели на которой планируется разместить заказ, артикул и наименование товара, количество, закупочная цена и сумма заказа.<http://localhost:8080/orders/5> - выводит заказ с id 5

Общую сумму заказов можно посмотреть по адресу <http://localhost:8080/orders/sum> (У нас оба поставщика работают в Евро, поэтому сумма выдана только в этой валюте)



Все полученные данные можно легко скопировать со страницы браузера в Word, Exсel или любой другой удобный офисный файл для сохранения результатов и дальнейшей работы.

**4. Выводы, планы по доработке и развитию проекта.**

Данный проект пока еще довольно сырой и находится в самой начальной стадии, далеко не все успел сделать из того, что хотелось. Можно сделать более удобный пользовательский интерфейс. Возможно даже придется немного пересмотреть логику подсчета заказов (эту логику я постарался максимально подробно расписать, но сейчас есть немного другие идеи). Есть также идеи в плане расширения и развития проекта. На данном этапе программа рассчитывает лишь прямые расходы на оплату заказов поставщикам.

Поэтому первым делом хотелось бы добавить расчеты сопутствующих транспортных и таможенных расходов. Сейчас даже упрощенный коэффициент deliveryCostFactor хоть и вводится, но в расчетах пока участия не принимает.

Возможно ввести дополнительные поля для товаров Item, например, дополнительное поле класса Package, где будут указаны вес, объем, уточнена единица измерения (штуки, килограммы, наборы). Возможно введение поля, отвечающего за таможенный код и величину пошлины. Все это в дальнейшем позволит уточнить расчеты и автоматизировать подготовку информации и документации для таможенного брокера или таможенного поста напрямую, и максимально облегчить работу специалиста по закупкам.

Можно даже будет настроить вывод заказа и шаблона письма поставщику для отправки по e-mail.

Более того, похожим способом можно также считать и доходную часть от продажи товаров и видеть общую картину ожидаемых доходов и расходов. Только видя общую картину, можно грамотно спланировать дальнейшее развитие бизнеса.

**Список использованных инструментов:** Spring Boot, Hibernate, H2Database, Thymeleaf, Apache POI

**Cписок использованной литературы:**

**-** Герберт Шилдс **«**JAVA 8 Руководство для начинающих[.» Издательский дом Видьямс, Москва 2018 г 706 стр](https://www.thymeleaf.org/" \t "_blank)

- Брюс Эккель «Философия Java» Издательство «Питер» 2022 г 1168 стр

- Марк Хеклер «SPRING BOOT по-быстрому» ООО «Прогресс книга» 2022 г. 349 стр

- материалы лекций и семинаров GeekBrains.