**Лекция: Практический проект «Листинг товаров» на микросервисной архитектуре**

## 1. Вводная часть: Микросервисы и их роль в проекте

Микросервисы – это подход к разработке, при котором приложение разбивается на набор мелких независимых сервисов, взаимодействующих через сеть (обычно по API)[kinsta.com](https://kinsta.com/blog/python-microservices/#:~:text=Microservices%20are%20small%2C%20independent%20applications,flexibility%20and%20manageability%20throughout%20development). В отличие от монолитного приложения, где весь функционал находится в одном коде, микросервисная архитектура подразумевает **несколько отдельных компонентов**, каждый из которых отвечает за свою бизнес-задачу. Такие сервисы слабо связаны между собой, что дает ряд преимуществ:

* **Гибкость и масштабируемость:** Каждый сервис можно развивать и масштабировать отдельно. Если нужна большая нагрузка на определённую функцию (например, каталог товаров), можно запустить больше экземпляров только соответствующего сервиса[kinsta.com](https://kinsta.com/blog/python-microservices/#:~:text=There%20are%20several%20benefits%20to,using%20microservices%2C%20such%20as). Это эффективнее, чем масштабировать весь монолит целиком.
* **Независимость технологий:** Разные микросервисы можно писать на разных языках программирования и использовать разные технологии, выбирая оптимальный инструмент для каждой задачи[kinsta.com](https://kinsta.com/blog/python-microservices/#:~:text=,development%20tools%20for%20each%20one). Например, в нашем проекте мы можем использовать Python для одних сервисов и PHP для других.
* **Устойчивость к сбоям:** Отказ одного микросервиса не приводит к падению всего приложения. Сервисы общаются через четко определённые интерфейсы (REST API), поэтому при проблеме в одном компоненте остальные продолжают работать. Это повышает надежность системы[kinsta.com](https://kinsta.com/blog/python-microservices/#:~:text=Unlike%20monolithic%C2%A0systems%20that%20scale%20entirely,and%20feature%20updates%2C%20countering%20monolithic%C2%A0limitations).
* **Ускорение развития командой:** В команде из нескольких человек микросервисная архитектура позволяет распределить работу – каждый может заниматься отдельным сервисом. Сервисы имеют чёткие границы ответственности, что уменьшает конфликт изменений в коде.

Важно отметить, что за гибкость приходится платить дополнительной сложностью – потребуется настроить взаимодействие сервисов, мониторинг нескольких компонентов и т.д. Для нашего учебного проекта эти сложности мы сведем к минимуму, сконцентрировавшись на базовых принципах.

**Применение микросервисов в проекте «Листинг товаров»:** Мы собираемся разработать систему каталогизации товаров, к которой затем добавим модуль пользователей, отзывы и корзину. Микросервисный подход позволит нам изолировать эти части. На практическом уровне это означает, что разные компоненты (товары, пользователи, заказы и пр.) будут реализованы как самостоятельные сервисы, которые взаимодействуют по HTTP API. Например, каталог товаров – отдельный сервис, управление пользователями – другой сервис и т.д. Это облегчит одновременную работу 3-5 новичков над проектом, так как каждый сможет сфокусироваться на своем участке системы.

 Пример микросервисной архитектуры: Клиентские приложения (браузер, мобильное приложение) обращаются к единой точке входа (API Gateway), который перенаправляет запросы к различным сервисам (например, аккаунты, инвентарь, доставка), каждый из которых имеет свою базу данных[medium.com](https://medium.com/@khouloud.haddad/php-microservices-without-frameworks-native-php-for-modern-architectures-8b8be104a7d8#:~:text=6). В нашем учебном проекте архитектура будет упрощенной – у нас меньше сервисов, но принцип разделения на независимые компоненты остаётся.

## 2. Планирование архитектуры проекта

Перед началом кодирования необходимо совместно спланировать архитектуру системы. Мы можем организовать **онлайн-митинг с электронной доской**, чтобы всем участникам было видно и можно было вносить правки. Например, сервис Miro Lite предоставляет бесплатную онлайн-доску, где можно совместно набрасывать схему архитектуры без регистрации[miro.com](https://miro.com/online-whiteboard/#:~:text=,a%20Miro%20Lite%20online%20whiteboard%3FImage). Такой инструмент позволит нам в реальном времени нарисовать схему сервисов и их взаимодействия (что особенно удобно, так как все участники находятся в разных местах).

**Предлагаемая архитектура:** Наш мини-каталог товаров изначально будет состоять из двух основных компонентов (микросервисов):

* **Сервис каталога товаров (Product Service)** – основной сервис, предоставляющий API для списка товаров. Мы планируем реализовать его на PHP (как более сложную часть, чтобы отработать PHP на практике). Этот сервис будет отвечать на HTTP-запросы (REST API) от фронтенда: выдавать список товаров, детали товара и т.п. В перспективе он может также обрабатывать запросы на добавление товаров в корзину или поиск, но на первом этапе достаточно реализации чтения списка.
* **Сервис управления данными/вспомогательный (Migration/Data Service)** – вспомогательный компонент на Python, который будет выполнять задачи по инициализации и обновлению базы данных. Его роль – запуск миграций (создание таблиц, заполнение первоначальными данными) и, потенциально, выполнение фоновых задач или интеграций. Пока этот сервис не будет постоянно запущен как веб-приложение; скорее, это набор Python-скриптов, которые команда запускает для изменения структуры базы или массового импорта данных. В будущем, когда мы добавим, скажем, модуль пользователей, можно сделать его на Python как отдельный микросервис.

**Взаимодействие компонентов:** Фронтенд (HTML/JS) будет обращаться к API каталога (PHP-сервису) через HTTP (например, GET /api/products). PHP-сервис обращается к базе данных для получения данных. Python-скрипты миграций также напрямую работают с базой данных, но не взаимодействуют с PHP-сервисом во время выполнения (они запускаются отдельно, при развёртывании или обновлении схемы БД). Таким образом, база данных выступает общей точкой, через которую косвенно связаны два наших компонента.

**Примечание:** В классической микросервисной архитектуре у каждого сервиса обычно своя собственная база данных для обеспечения слабой связанности[medium.com](https://medium.com/@khouloud.haddad/php-microservices-without-frameworks-native-php-for-modern-architectures-8b8be104a7d8#:~:text=6). Мы должны понимать этот принцип, однако для упрощения учебного проекта можем использовать одну общую базу для всех модулей (чтобы не усложнять развёртывание и поддержку). Если получится, можно логически разделить данные по схемам или именам таблиц, относящимся к разным сервисам.

**Онлайн-инструменты для проектирования архитектуры:** Помимо Miro, существуют и другие бесплатные коллаборативные доски (Excalidraw, Ziteboard и т.д.)[miro.com](https://miro.com/online-whiteboard/#:~:text=Online%20Whiteboard%20for%20Realtime%20Collaboration,up%20or%20registration%20is%20required)[ziteboard.com](https://ziteboard.com/whiteboard-collaboration-tool/#:~:text=Ziteboard%20ziteboard,many%20other%20types%20of). Главное – выбрать то, что доступно в России и не требует оплаты. Miro Lite хорош тем, что не требует даже аккаунта: достаточно открыть ссылку и все участники могут рисовать одновременно. На этой доске мы изобразим компоненты (Frontend, Product API на PHP, Migration Service на Python, Database) и стрелками покажем, как идут запросы и данные между ними. Это будет черновая схема архитектуры, которая обеспечит общее понимание у всей команды.

## 3. Планирование базы данных

После архитектуры на высоком уровне, следующим шагом станет проектирование базы данных. Правильная структура данных – фундамент любого проекта, особенно каталога товаров. Совместное создание **ER-диаграммы (диаграммы «сущность-связь»)** поможет всем участникам понять, какие таблицы нужны и как они связаны[owox.com](https://www.owox.com/blog/articles/database-diagram-design-tools#:~:text=Designing%20a%20clear%20and%20organized,foundation%20for%20your%20data%20projects). Мы можем использовать онлайн-инструмент вроде dbdiagram.io для этого – он бесплатный и позволяет описывать схему таблиц в виде простого текста, генерируя наглядную диаграмму[owox.com](https://www.owox.com/blog/articles/database-diagram-design-tools#:~:text=1). Такой подход не требует установки ПО, достаточно браузера, и результат легко шарится через ссылку.

**База данных для листинга товаров – что нужно:**

* Начнём с таблицы **products** (товары). В ней могут быть поля: id (первичный ключ, автоинкремент), name (название товара), description (описание), price (цена), image\_url (ссылка на изображение) и др. – по необходимости. Минимально для листинга нам хватит названия и цены, но добавим описание для пользы.
* Возможна таблица **categories** (категории) с полями id, name, чтобы сгруппировать товары. Если будем использовать, то в products добавим поле category\_id (внешний ключ на categories). Это пригодится, если хотим фильтровать каталог по категориям.
* В перспективе (но не прямо сейчас) понадобятся таблицы для пользователей (users), отзывов (reviews или часть таблицы товаров), заказов/корзины (orders, order\_items). Пока мы их можем просто наметить, но не реализовывать, либо заложить возможность добавления.
* Если планируется хранить пароли пользователей, учитывать безопасность (хранить хеши паролей, и т.д.), но это, вероятно, выйдет за рамки текущей сессии. На данном этапе мы можем отложить users и другие, сосредоточившись на **товарах**.

**Совместное онлайн-планирование БД:** Можно организовать мозговой штурм по дизайну БД аналогично архитектуре – на онлайн-доске или прямо в dbdiagram.io совместно. Например, каждый предлагает, какие поля нужны, обсуждаем связи. Инструменты для коллективного моделирования БД, поддерживающие реальное время, менее распространены, но мы можем по очереди редактировать диаграмму. Еще вариант – использовать Google Sheets или Excalidraw, чтобы вручную нарисовать таблицы и связи. Главное, чтобы все участники пришли к единому решению.

**Миграции базы данных:** Поскольку мы предполагаем, что структура БД будет меняться и дополняться по мере развития проекта, необходимо заложить механизм миграций. **Миграция** – это изменение структуры БД (создание/изменение таблиц, добавление столбцов и т.д.), оформленное как отдельный шаг, который можно воспроизвести автоматически на любой базе. Использование миграций дает контроль версий схемы, позволяет откатывать изменения и синхронизировать базы между разными окружениями[dev.to](https://dev.to/r0mymendez/simplify-database-migrations-using-python-with-alembic-4bhd#:~:text=,maintaining%20control%20over%20schema%20versions). В профессиональной среде есть инструменты миграций (в мире Python – Alembic[dev.to](https://dev.to/r0mymendez/simplify-database-migrations-using-python-with-alembic-4bhd#:~:text=What%20is%20Alembic%3F), в PHP-фреймворках типа Laravel – встроенные миграции, и др.). Мы же реализуем упрощённый свой подход с помощью Python-скриптов:

* Создадим папку, например, migrations/, где каждая миграция будет отдельным пронумерованным скриптом (например, 001\_create\_products.py, 002\_add\_users.py и т.д.). Нумерация или дата в имени поможет определить порядок их выполнения.
* Первая миграция **создаст начальные таблицы**: products, возможно categories, и наполнит таблицу products некоторыми тестовыми данными (несколько товаров с названиями и ценами), чтобы сразу было что выводить в каталоге.
* Будущие миграции будут добавлять новые таблицы (например, таблицу пользователей) или изменять существующие (например, добавить колонку "rating" в products или что-то ещё).
* Мы напишем также скрипт или инструкцию по выполнению миграций: например, небольшой Python-скрипт apply\_migrations.py, который будет подключаться к базе и по порядку выполнять все еще не выполненные миграции. Для простоты можно миграции применять вручную (запуская скрипты по порядку), но лучше автоматизировать. Возможно, просто будем отслеживать, какая последняя миграция применена (например, в файле или в специальной таблице schema\_version в БД).

## 4. Среда разработки: совместный онлайн-кодинг и работа с GitHub

Поскольку у нас командная работа, важно решить, как несколько человек будут вместе писать код. Есть два подхода:

* **Локальная разработка с использованием Git** – каждый работает на своем компьютере (это приоритетный вариант, как вы отметили), а код объединяется через общий репозиторий на GitHub. Здесь важно заранее распланировать, кто за что отвечает, чтобы минимизировать конфликты. Например, один пишет фронтенд, другой – бэкенд на PHP, третий – Python-скрипты. GitHub позволит сливать изменения, но одновременно редактировать один и тот же файл не очень удобно (возникнут merge conflict’ы, с которыми новичкам трудно).
* **Онлайн IDE с коллаборацией в реальном времени** – все работают в одном онлайн-редакторе, видя правки друг друга мгновенно. Это похоже на Google Docs, но для кода. Современные онлайн-IDE предоставляют такую возможность[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=2,and%20in%20most%20educational%20setups). Преимущество – моментально видно, кто что пишет, можно помогать друг другу; недостаток – требуется хорошее интернет-соединение, и иногда такие сервисы ограничены по функционалу или доступности.

Чтобы совместить плюсы, можно делать так: используем **онлайн-IDE как временное рабочее пространство** для совместного творчества, а результат сохраняем в GitHub-репозитории. Нужно подобрать сервис, который **бесплатен, популярен и доступен в России** (с учетом возможных ограничений). Рассмотрим несколько вариантов:

* **Replit (repl.it):** Очень популярная платформа для мульти-языковой разработки прямо в браузере. Поддерживает более 50 языков (включая Python, PHP, JS и многое другое)[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/" \l ":~:text=Repl,such%20as%20Node%2C%20Python%2C%20C" \t "_blank). Replit позволяет нескольким пользователям редактировать один проект одновременно (реальное время) и запускать код. Есть возможность импортировать проект с GitHub и обратно синхронизировать[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=from%20a%20GitHub%20repository,GitHub%2C%20Apple%2C%20or%20Facebook%20account). Бесплатный план довольно щедрый (до 1000 часов работы в месяц, что для наших нужд достаточно)[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/" \l ":~:text=Pros%20of%20using%20Repl" \t "_blank). У Replit активное сообщество, множество примеров. Поскольку у нас бэкенды на разных языках, возможно, имеет смысл создать два отдельных проекта (один для PHP, другой для Python), или попытаться уложить их в один репл. Например, можно создать в Replit проект типа "PHP Web Server" для API, а Python-скрипты там же хранить и запускать через встроенный консольный доступ. Либо параллельно запустить два репла и подключиться к каждому.
* **CodeSandbox:** Отлично подходит для веб-разработки (особенно фронтенда). Поддерживает совместное редактирование и интеграцию с GitHub. CodeSandbox наиболее гибкий из онлайн-редакторов с точки зрения структуры проектов[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=match%20at%20L695%20end%20projects,most%20flexible%20when%20creating%20projects). Однако, основной упор там на Node.js и фронтенд. PHP напрямую может не поддерживаться, хотя есть шаблоны с контейнером. Если бы весь бэкенд был на Node, это был бы идеальный вариант. Для нашего случая – может быть полезен для фронтенда (HTML/CSS/JS) – команда могла бы верстать страницу в CodeSandbox, а API вызывать с локального хоста.
* **GitHub Codespaces:** Интегрированная среда разработки от GitHub. Прямо в репозитории можно открыть онлайн-IDE (VS Code в браузере) со всем кодом. Плюс – непосредственная связь с репозиторием, минус – ограниченные бесплатные ресурсы (по состоянию на 2025 год дается несколько десятков часов в месяц бесплатно, нужно экономно использовать). Также, непонятно, нет ли ограничений для российских пользователей. Но технически это очень мощный вариант, близкий к полноценной VS Code.
* **Другие:** StackBlitz (хорош для JS/Node, позволяет тоже GitHub-проекты открывать[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=Additionally%2C%20you%20can%20start%20a,stack%20applications)), Glitch (похож на Replit, но фокус на Node.js; поддерживает совместное редактирование, бесплатный), Visual Studio Live Share (расширение для VS Code, позволяет пригласить других в свою сессию; хорошее решение, если все могут установить VS Code и есть стабильный канал связи – тогда вся работа происходит на одном из компьютеров, а остальные как «клиенты» редактируют в реальном времени).

Учитывая ваши критерии ("популярное и бесплатное"), **Replit** выглядит наиболее подходящим компромиссом. Его веб-интерфейс прост, многие новички уже знакомы с ним. Import/Export с GitHub присутствует, так что можно начать кодить там, а потом запушить результат в репозиторий[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=How%20to%20import%20GitHub%20project,it). Также Replit не требователен к компьютеру участников – вся тяжелая работа происходит на сервере (подходит, если у кого-то слабый ноутбук)[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/" \l ":~:text=1,order%20to%20make%20coding%20accessible" \t "_blank).

**Рекомендации по совместной работе:**

* Если решаем работать локально: разделите зоны ответственности и регулярно сливайте изменения через Git. Например, один участник создает базовую структуру PHP-проекта, другой – пишет скрипты миграций, третий – каркас фронтенда. Каждый форкает/клонирует репозиторий, работает и пушит изменения. Конфликты в Git можно минимизировать, если сразу оговорить, кто какие файлы редактирует. Например, файл index.php API трогает только 1 человек или согласованно.
* Если пробуем онлайн-IDE: соберитесь все одновременно в выбранном инструменте, разделите там экран (многие IDE показывают аватарки, кто где курсор держит) – например, договоритесь, что Аня редактирует файл index.html, Борис пишет в products.php, etc. Обязательно периодически сохраняйте/экспортируйте код в локальные копии или в репозиторий, чтобы ничего не потерять (хотя большинство онлайн-IDE авто-сохраняют код и имеют бэкапы[refine.dev](https://refine.dev/blog/6-best-online-code-editors-comparison/#:~:text=5,the%20danger%20of%20losing%20information)).
* **Компромиссный вариант:** Использовать онлайн-IDE в режиме демонстрации/проверки, а основную работу выполнять локально. Например, лектор или тимлид может открыть проект в Replit на общем экране (в режиме Only View для остальных) – и вместе с группой просматривать код, запускать, тестировать. А правки каждый будет делать у себя и отправлять через GitHub. Но это сложнее синхронизировать. Все зависит от уровня владения Git у участников.