СИСТЕМНЫЙ ДИЗАЙН ДЛЯ ВЕБ-ПРИЛОЖЕНИЯ ДЛЯ ПРАКТИКИ РЕШЕНИЯ АЛГОРИТМИЧЕСКИХ ЗАДАЧ

Тишалович М.А.

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Потапов В.Д. – канд. техн. наук, доцент

Разработан системный дизайн для веб-приложения для практики решения алгоритмических задач, обеспечивающий высокую производительность и возможность горизонтального масштабирования. Предложено использование микросервисной архитектуры, протокола WebSockets, брокера сообщений Apache Kafka для решения проблем, с которыми сталкиваются программисты при разработке таких приложений.

В простейшем виде подобная система будет являться веб-приложением с клиент-серверной архитектурой, где клиентское приложение представляет собой SPA, а сервер предоставляет клиенту REST API. SPA – это реализация веб-приложения, которая загружает только один веб-документ, а затем обновляет содержимое тела этого одного документа через API JavaScript, такие как Fetch, когда необходимо отобразить другой контент [1]. REST API – это архитектурный стиль для создания веб-сервисов, основанный на стандартных HTTP-методах и принципах REST [2]. Однако, такая архитектура имеет ряд недостатков. Так как проверка решения может занимать много времени, может произойти такая ситуация, что веб-клиент не дождется ответа и, например, получит request-timeout.

Для того, чтобы избежать проблем подобного рода, необходимо отправить клиенту сообщение, что решение принято в обработку, после чего, когда решение было проверено, сервер должен отправить клиенту сообщение с результатом проверки решения задачи. Для реализации такого подхода можно использовать, например, Long Polling или WebSockets. Long polling – это техника, используемая в веб-разработке для эмуляции связи в реальном времени между клиентом (обычно веб-браузером) и сервером. В отличие от обычного HTTP-запроса, где клиент ждет ответа сразу, long polling позволяет серверу держать соединение открытым до тех пор, пока не появятся новые данные или не истечет время ожидания [3]. WebSockets – это протокол связи, который обеспечивает двустороннее (full-duplex) взаимодействие между клиентом и сервером через одно постоянное соединение. В отличие от HTTP, где клиент инициирует запросы, а сервер отвечает, WebSockets позволяют обеим сторонам отправлять данные в любой момент без необходимости повторного установления соединения [4]. WebSockets – является более современной технологией, чем Long polling, причем сообщения в технологии WebSockets доставляются мгновенно, так как соединение постоянно открыто, в отличие от задержек в Long polling при периодических запросах. Поэтому для данной системы было решено использовать WebSockets. Таким образом, для отправки сообщения клиент использует WebSockets для общения с сервером, а для остальных целей общается с сервером через REST API.

Однако, стоит заметить, что в таком случае не решается проблема с перегруженностью сервера, так как ему в любом случае нужно будет тратить много ресурсов на проверку решений. Для того, чтобы сервер, с которым напрямую взаимодействуют клиенты не перегружался, было решено использовать отдельные сервисы, которые будут заниматься проверкой решений задач. В таком случае также появляются проблемы, связанные с тем, как сервису, с которым общается клиент, передавать решения сервисам проверки. Одним из решений может быть отдельная база данных, в которой будут складываться решения, которые необходимо проверить, а сервисы для проверки решений будут вытягивать из неё эти решения. Однако, в таком случае необходимо полностью разработать логику по распределению решений между сервисами по проверке решений. Вместо этого возможно использовать готовое решение – брокер сообщений, который предоставляет собой производительный способ для передачи сообщений от отправителя к получателям. Брокер сообщений – это программное обеспечение, которое позволяет приложениям, системам и службам взаимодействовать друг с другом и обмениваться информацией [5]. Одним из таких брокеров сообщений является Apache Kafka. Apache Kafka – это распределённая платформа для потоковой обработки данных, разработанная для высокопроизводительного, отказоустойчивого и масштабируемого обмена сообщениями между системами [6]. С помощью данного брокера сервис, взаимодействующий с клиентом, будет только получать решения и сразу же добавлять их в очередь, откуда сервисы по проверке решений будут их вынимать тогда, когда будут свободны. Данное решение позволяет убрать нагрузку с сервера, который взаимодействует с клиентами, так и позволить системе расширяться горизонтально, так как количество сервисов, занимающихся проверкой решений, можно варьировать в зависимости от того, какая нагрузка у платформы.

Стоит заметить, что в данном решении сервер, с которым взаимодействуют клиенты, выполняет много различных функций, как обработка отправленных решений по WebSockets, так и получение всей остальной информации через REST API. Это подвергает данный сервер лишней нагрузке. Этой

проблемы можно избежать, разбив данный сервер на отдельные сервисы, каждый из которых будет заниматься только определённой задачей. Это не только увеличит производительность, но также и устойчивость, так как остальные сервисы будут продолжать работать даже в том случае, если какой-то из них станет недоступен. Следующим недостатком данной системы является единственная база данных. Это решение является плохим как с точки зрения производительности, так и безопасности. Для устранения данных проблем было решено выделить отдельную базу данных для каждого сервиса. Причем, с точки зрения безопасности необходимо создать отдельную базу данных для сервисов по проверке решений. Это позволит обеспечить уверенность в том, что тесты, на которых проверяются задачи, не будут получены злоумышленниками или же случайно отправлены клиенту. Однако, при появлении множества сервисов, появляется проблема того, что клиенту нужно обращаться ко всем сервисам отдельно, что не является удобным. Для решения этой проблемы было решено использовать паттерн API Gateway. API Gateway представляет собой шлюз, который является единой точкой входа для всех клиентов. Данный шлюз обрабатывает запросы одним из двух способов: некоторые запросы просто проксируются/маршрутизируются в соответствующую службу, а другие запросы он обрабатывает, разветвляясь на несколько служб [7]. Полученная система изображена на рисунке 1.

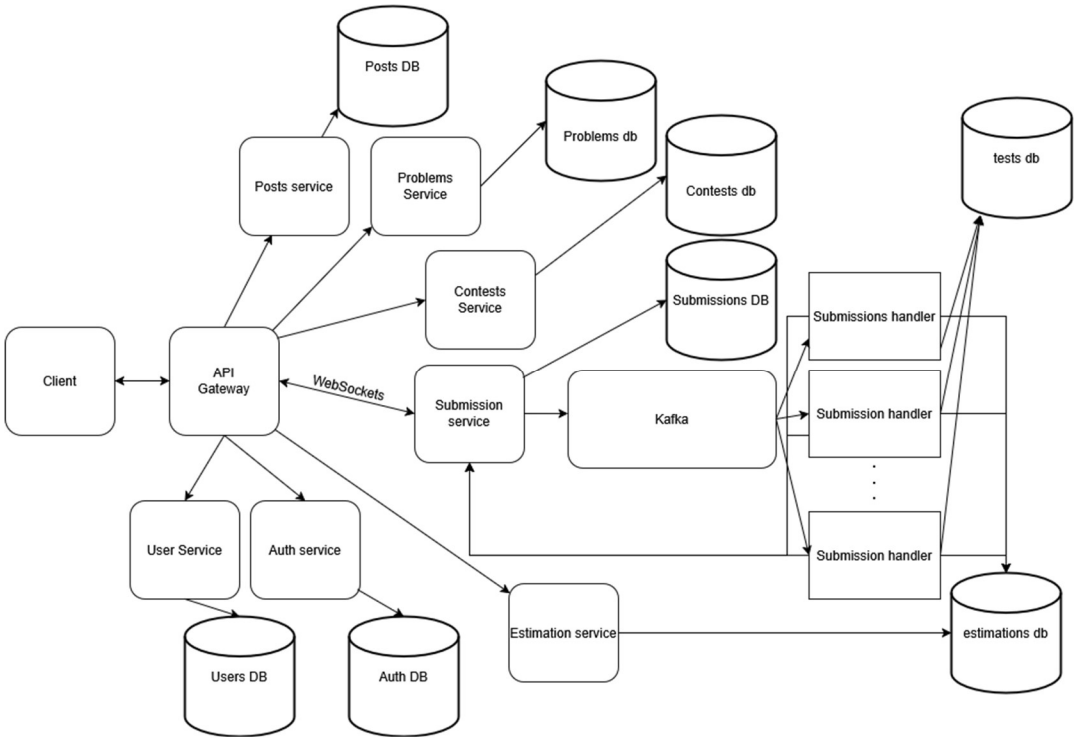

Рисунок 1 – Финальная версия системного дизайна

В результате разработки системного дизайна предложено использование микросервисной архитектуры для серверной части приложения, где используется REST API для получения информации о задачах, турнирах, пользователях, постах и используется протокол WebSockets для отправки решений задачи. Также было предложено использование брокера сообщений Apache Kafka для распределения отправленных решений между микросервисами проверки решений.

Список использованных источников:

1. Mozilla SPA [Электронный ресурс]. – Режим доступа : https://developer.mozilla.org/en-US/docs/Glossary/SPA. – Дата доступа: 31.03.2025.

2. Amazon RESTful API [Электронный ресурс]. – Режим доступа : https://aws.amazon.com/ru/what-is/restful-api/. – Дата доступа: 31.03.2025.

3. Pub num Long Polling [Электронный ресурс]. – Режим доступа : https://www.pubnub.com/guides/long-polling/. – Дата доступа: 31.03.2025.

4. MDN WebSockets [Электронный ресурс]. – Режим доступа : https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. – Дата доступа: 31.03.2025.

5. IBM Message Broker [Электронный ресурс]. – Режим доступа : https://www.ibm.com/think/topics/message-brokers. – Дата доступа: 31.03.2025.

6. Apache Kafka [Электронный ресурс]. – Режим доступа : https://kafka.apache.org/. – Дата доступа: 31.03.2025.

7. Microservices IO API Gateway [Электронный ресурс]. – Режим доступа : https://microservices.io/patterns/apigateway.html. – Дата доступа: 31.03.2025.