1 Зачем нужна реактивность

Прежде всего мы хотим построить *отзывчивую* систему. Это означает, что система должна своевременно реагировать на действия пользователя в любых условиях. Каждый отдельный компьютер в любой момент времени подвержен угрозе неисправности, поэтому необходимо распределить эту систему между несколькими компьютерами. Делая распределенность фундаментальным требованием, мы сталкиваемся с необходимостью новых (или давно забытых старых) шаблонов проектирования. Когда-то была разработана методика, позволявшая сохранять видимость локальных однопоточных вычислений, которые на самом деле магическим образом выполнялись на нескольких ядрах или сетевых узлах, однако расхождение между этой иллюзией и реальностью становится все заметнее[1]. Чтобы решить эту проблему, распределенную конкурентную сущность наших приложений следует явно отразить в модели программирования и использовать в качестве преимущества.

Эта книга научит вас создавать системы, которые остаются отзывчивыми при кратком отключении электричества, программных сбоях, колеблющихся нагрузках и даже ошибках в коде. Вы увидите, что для этого необходимо по-новому взглянуть на свои приложения и изменить способ их проектирования. Далее приведены четыре принципа манифеста реактивного программирования[2], определяющего общую терминологию и описывающего основные вызовы, к которым должна быть готова современная компьютерная система.

- Она должна реагировать на действия своих пользователей — *отзывчивость*.
- Она должна реагировать на сбои и оставаться доступной — *устойчивость*.
- Она должна реагировать на колебания нагрузки — *эластичность*.
- Она должна реагировать на ввод — *ориентированность на обмен сообщениями*.

Кроме того, создание системы на основе перечисленных свойств научит вас делать развернутые приложения и сам код более модульными. Таким образом,

---

[1] Например, сервисы Java EE позволяют прозрачно вызывать удаленный код, который выполняется автоматически и может даже включать в себя транзакции в распределенных базах данных. Вероятность сетевого сбоя или перегрузки службы полностью скрыта за абстракциями, из-за чего у разработчиков нет возможности учитывать их.

[2] См.: reactivemanifesto.org/.

к списку положительных характеристик можно добавить еще два пункта: *поддерживаемость* и *расширяемость*. Один из способов представления этих атрибутов показан на рис. 1.1.

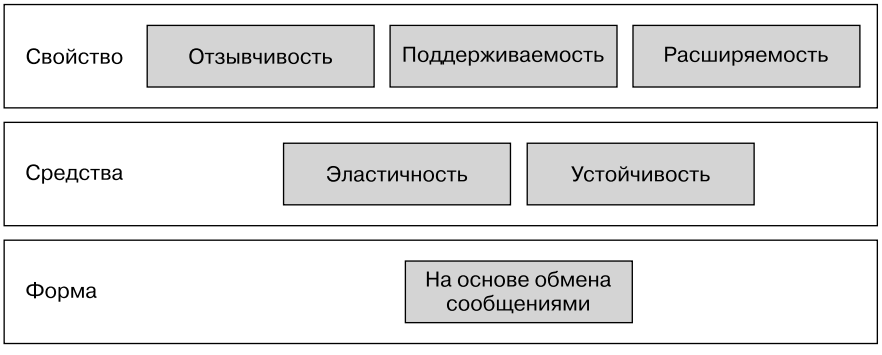

Рис. 1.1. Структура реактивных свойств

В следующих главах мы подробнее расскажем о критериях, на которых основан манифест реактивного программирования, а также опишем несколько подходящих инструментов и философию, лежащую в их основе, что позволит вам эффективно использовать эти инструменты для реализации реактивных архитектур. Шаблоны проектирования, появившиеся благодаря данным инструментам, представлены в части III книги. Но прежде, чем погружаться в сам манифест, мы рассмотрим проблемы, которые стоят на пути создания реактивного приложения. Для этого в качестве примера возьмем всем известный почтовый сервис Gmail и попробуем представить себе его альтернативную реализацию.

1.1. Анатомия реактивного приложения

Приступая к подобному проекту, первым делом следует набросать архитектуру процесса развертывания и составить список программных компонентов, которые нужно разработать. Эта архитектура не обязательно должна быть окончательной, но вам необходимо определить проблематику и наметить аспекты, с которыми могут возникнуть трудности. Мы начнем работать над примером с Gmail, перечислив высокоуровневые характеристики приложения.

- Приложение должно предлагать пользователю просмотр почтовых ящиков и выводить их содержимое.
- Для этого система должна хранить все письма и обеспечивать их доступность.
- Пользователь должен иметь возможность составлять и отправлять письма.
- Чтобы сделать процесс более комфортным, система должна предлагать список контактов и позволять пользователю ими управлять.
- Требуется также хорошая поисковая функция для нахождения писем.

Настоящее приложение Gmail имеет больше возможностей, но для наших целей этого списка вполне достаточно. Некоторые из данных пунктов пересекаются между собой больше, чем другие: например, отображение писем и их составление являются частью пользовательского интерфейса и занимают (или претендуют на это) одно и то же место на экране, тогда как реализация хранилища писем имеет к ним опосредованное отношение. Поисковой функции потребуется быть ближе к хранилищу, нежели к представлению на клиентской стороне.

По изложенным соображениям функциональность Gmail лучше иерархически разбить на все меньшие и меньшие составляющие. В частности, вы можете применить *шаблон «Простой компонент»*, как описано в главе 12, чтобы четко разделить и изолировать разные зоны ответственности в рамках всего приложения. Этот процесс дополняется *шаблонами «Ядро ошибок»* и *«Допустимый отказ»*, которые подготавливают приложение к надежной обработке ошибок не только в ситуациях с аппаратными или сетевыми неполадками, но и в тех редких случаях, когда исходный код некорректно обрабатывает сбой (то есть в случае *программных ошибок*, или *багов*).

Результатом этого процесса станет иерархия компонентов, которые нужно разработать и развернуть. Пример показан на рис. 1.2. Каждый компонент может быть сложным с точки зрения своей функциональности, как в случае с реализацией поисковых алгоритмов, или в плане развертывания и управления, как это будет с хранилищем писем для миллиардов пользователей. Но описание области ответственности данных компонентов всегда будет простым.

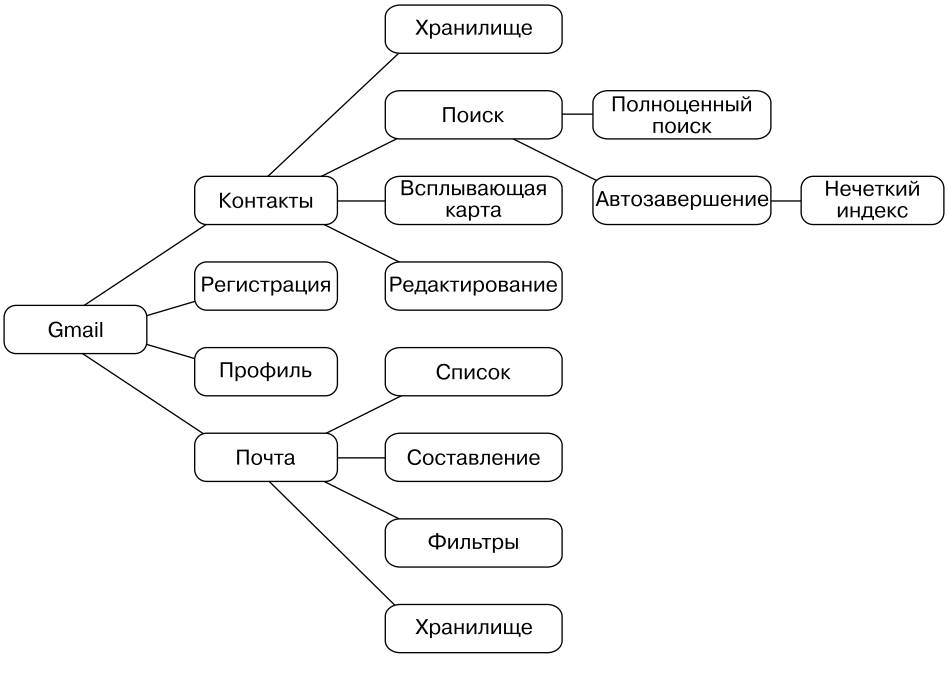

Рис. 1.2. Частично модульная иерархия гипотетической реализации Gmail

1.2. Справляемся с нагрузкой

Для хранения всех этих писем необходимы огромные ресурсы: хранилище должно вмещать гигабайты данных каждого пользователя, на что потребуются эксабайты[1] свободного места. Постоянное хранилище такого объема должно быть составлено из множества распределенных компьютеров. Не существует такого устройства, которое могло бы самостоятельно предоставить столько дискового пространства, к тому же хранить все в одном месте было бы неразумно. Распределение делает данные устойчивыми к локальным рискам, таким как природные катастрофы, но, что более важно, это позволяет также обеспечивать эффективный доступ к данным на большой территории. Если ваши пользователи разбросаны по всему миру, данные тоже должны быть распределены глобально. Хранить письма японского пользователя предпочтительнее в самой Японии или поблизости (в случае если пользователь чаще всего входит в систему из этого региона).

Это приводит нас к *шаблону сегментирования*, описанному в главе 17: набор данных можно разбить на множество мелких частей — *сегментов*, которые затем распределить. Поскольку количество сегментов намного меньше количества пользователей, местонахождение каждого сегмента имеет смысл сделать известным всей системе. Чтобы найти почтовый ящик пользователя, достаточно идентифицировать сегмент, к которому он принадлежит. Для этого каждому пользователю можно присвоить идентификатор, имеющий географическую привязку (например, несколько начальных цифр будут обозначать страну проживания) и математически распределенный на подходящее количество сегментов (например, сегмент 0 содержит идентификаторы 0–999 999, сегмент 1 — идентификаторы 1 000 000–1 999 999 и т. д.).

Ключевой особенностью здесь является то, что набор данных естественным образом состоит из независимых частей, которые можно легко отделить друг от друга. Операции с одним почтовым ящиком не влияют непосредственно на другие почтовые ящики, поэтому сегментам не нужно взаимодействовать между собой. Каждый из них является какой-то одной определенной частью решения.

Еще одной возможностью приложения Gmail, которая потребует много ресурсов, является вывод пользователю папок и писем. Подобную функцию невозможно предоставить централизованным способом — не только из-за задержек (даже на скорости света информации нужно значительное время, чтобы обогнуть земной шар), но и из-за самого количества действий, выполняемых миллионами пользователей ежесекундно. Кроме того, мы распределим работу между множеством компьютеров, начиная с принадлежащих самим пользователям: географическое представление по большей части формируется в рамках браузера, что смещает нагрузку поближе к тому месту, где она требуется, и в результате сегментирует ее для каждого пользователя.

Браузеру нужно будет получить сырую информацию от сервера — в идеале от находящегося поблизости, чтобы минимизировать время передачи пакетов туда и обратно. Подключение пользователя к его почтовому ящику и соответствующая

[1] 1 Эбайт [Eb] = 1 млрд гигабайт [Gb] (в десятичном счислении, в двоичной системе 1 Эбайт ≈ 1,07 млрд гигабайт).

маршрутизация запросов/ответов тоже легко сегментируются. В данном случае сетевой адрес браузера непосредственно предоставляет все необходимые сведения, включая географическое местоположение.

Один из аспектов, на который следует обратить внимание, заключается в том, что во всех вышеупомянутых сценариях ресурсы можно добавлять, уменьшая размер сегментов и распределяя нагрузку между бóльшим количеством компьютеров. Максимальное их число определяется исходя из количества пользователей или задействованных сетевых адресов, чего будет более чем достаточно для предоставления достаточного объема ресурсов. Этот подход нужно корректировать, только если обслуживание одного пользователя требует ресурсов, которые нельзя обеспечить с помощью одного компьютера. В такой ситуации пользовательский набор данных или вычислительную задачу следует разбить на более мелкие части.

Это означает, что, разбивая систему на распределяемые части, вы получаете возможность масштабировать ее производительность, увеличивая количество сегментов для обслуживания пользователей. Теоретически, если сегменты не зависят друг от друга, систему можно масштабировать бесконечно. На практике же комбинирование и развертывание приложения на миллионы узлов по всему миру требуют существенных затрат и должны быть оправданными.

1.3. Обработка сбоев

Сегментирование наборов данных или вычислительных мощностей решает проблему предоставления достаточного количества ресурсов в идеальном случае, когда все работает гладко, а сеть всегда доступна. Чтобы быть готовыми к сбоям, вы должны иметь возможность продолжать работу даже после того, как что-то пошло не так.

- Компьютер может сломаться временно (например, в результате перегрева или сбоя ядра) или навсегда (при электрических или механических неполадках, в результате пожара, наводнения и т. д.).
- Могут отказать сетевые компоненты — как внутри вычислительного центра, так и снаружи, где-нибудь в Интернете (например, при повреждении межконтинентальных подводных кабелей, что приводит к потере связи с отдельными регионами Сети).
- Операторы-люди или автоматические обслуживающие скрипты могут случайно уничтожить часть данных.

Единственным решением этой проблемы является хранение нескольких копий системы, ее данных или функциональности в разных местах (это называется *репликацией*). Географическое размещение копий *(реплик)* должно соответствовать масштабам системы, например, глобальный почтовый сервис должен обслуживать всех клиентов из нескольких стран.

По сравнению с сегментацией репликация является более сложной и многогранной темой. Интуитивно нам хочется иметь одни и те же данные в нескольких местах, однако синхронизация реплик, которая при этом подразумевается, влечет за собой большие затраты. Что делать с записью в ближайшую реплику, если

более удаленная реплика стала на мгновение недоступной, — прервать, отложить? Следует ли сделать невозможным просмотр старых данных на удаленной реплике после того, как ближайшая копия уже сообщила о завершении операции? Или, может, просто стоит рассматривать такую несогласованность как маловероятную или очень кратковременную? Ответы на эти вопросы могут варьироваться для разных проектов или даже разных модулей в рамках одной системы. Таким образом, в вашем распоряжении оказывается целый спектр решений, который позволяет находить компромисс между сложностью выполнения, производительностью, доступностью и согласованностью.

В главе 13 мы обсудим несколько подходов, которые охватывают широкий диапазон характеристик. Далее перечислены основные варианты.

- *Репликация «Активный к пассивному»*. Реплики согласуют между собой, какая из них может принимать обновления. Когда активная реплика больше не отвечает, аварийное переключение на другую реплику требует согласия остальных копий.
- *Согласуемая репликация с несколькими активными копиями*. Каждое обновление утверждается определенным числом реплик, достаточным для достижения согласованного поведения в каждой из них. При этом страдают доступность и время отклика.
- *Оптимистичная репликация с обнаружением и разрешением конфликтов*. Множественные активные реплики распространяют обновления и откатывают транзакции в случае конфликтов или отменяют конфликтующие изменения, выполненные во время нарушения связанности сети.
- *Бесконфликтные реплицируемые типы данных*. Данный подход описывает такие стратегии слияния, при которых конфликты невозможны по определению. Это происходит за счет предоставления лишь так называемой *согласованности в конечном счете* и требует отдельных действий при создании моделей данных.

В примере с Gmail согласованность должны обеспечивать несколько сервисов: при успешном перемещении письма в другую папку пользователь ожидает, что оно находится в этой папке вне зависимости от клиента, который применяется для доступа к почтовому ящику. То же самое касается изменения телефонных номеров контактов или пользовательских профилей. Для этих данных можно использовать репликацию «Активный к пассивному», что позволит избежать усложнения системы благодаря более обобщенной обработке сбоев, которая действует в рамках отдельной реплики. Здесь также можно применить оптимистичную репликацию, исходя из того, что пользователь не станет одновременно вносить противоречивые изменения в один и тот же элемент данных (при этом необходимо понимать, что такое допущение справедливо только для реальных пользователей).

Согласуемая репликация нужна в системах в качестве элемента реализации сегментирования по пользовательскому идентификатору, так как запись о перемещении сегмента должна быть сделана корректно и согласованно для всех клиентов. Это приводит к искажениям, которые видны пользователю, — например, письмо может исчезнуть и затем снова появиться, если клиент перескакивал между просроченной и актуальной репликами.

1.4. Придание системе отзывчивости

Из предыдущих двух разделов вы узнали, какие причины стоят за распределением системы между несколькими компьютерами, вычислительными центрами или даже континентами — в зависимости от масштабов приложения и требований к его надежности. Первоочередной целью данного упражнения является создание почтового сервиса для конечных пользователей, а для них важно одно — делает ли приложение то, что им нужно и когда нужно. Иными словами, приложение должно быстро отвечать на любой запрос, сделанный пользователем.

Очевидно, проще всего этого добиться путем написания приложения, которое работает локально и хранит все письма на компьютере пользователя: получение ответа по сети всегда занимает больше времени и является менее надежным, чем в случае, когда ответ находится рядом. Таким образом, возникает противоречие между необходимостью распределения и потребностью в отзывчивости. Распределение всегда должно быть оправданным, как в случае с Gmail.

В ситуациях, когда распределение необходимо, в погоне за отзывчивостью мы сталкиваемся с новыми вызовами. Наиболее раздражающим аспектом многих распределенных систем сейчас является тот факт, что при плохом сетевом соединении взаимодействие с ними становится затрудненным. Забавно, что справляться с полным отсутствием соединения оказывается проще, чем с медленной передачей данных. Здесь может помочь *шаблон «Предохранитель»*, подробно описанный в главе 12, — с его помощью можно отслеживать доступность и производительность сервиса, который вызывается для выполнения какой-то функции. Когда качество падает ниже допустимого значения (либо слишком много сбоев, либо слишком большое время отклика), срабатывает предохранитель, заставляя систему переключиться в режим, в котором этот сервис не задействуется. Потенциальную недоступность некоторых участков системы следует учитывать с самого начала, и шаблон «предохранитель» решает эту проблему.

Еще одна угроза отзывчивости возникает, когда сервис, на который полагается приложение, в какой-то момент оказывается перегруженным. Начнут накапливаться отложенные запросы, и, пока они обрабатываются, ответы будут приходить со значительно большей задержкой, чем обычно. Этого можно избежать, прибегнув к *управлению потоком* (описывается в главе 16). В примере с Gmail есть несколько участков, которые требуют применения предохранителей и управления потоком:

- между клиентской стороной, которая выполняется на устройстве пользователя, и веб-серверами, предоставляющими доступ к серверным функциям;
- между веб-серверами и серверными сервисами.

Мы уже упоминали причину, стоящую за первым пунктом: желание сохранить отзывчивость доступной пользователю части приложения при любых условиях, даже если единственное, что она может сделать, — это сообщить о недоступности сервера и о том, что запрос будет выполнен позже. В зависимости от того, какой объем функциональности дублируется на клиентской стороне для реализации такого *автономного режима*, некоторые участки пользовательского интерфейса, возможно, придется деактивировать.

Второй пункт объясняется тем, что в противном случае клиентская часть должна была бы иметь разные предохранители для разных видов запросов к веб-серверу, при этом каждый предохранитель должен был бы отвечать за определенное подмножество серверных сервисов, необходимых для выполнения запроса определенного типа. Переключение всего приложения в автономный режим, когда лишь небольшая часть серверных сервисов является недоступной, было бы бесполезным и чрезмерным. Отслеживание доступности на клиентской стороне привязало бы ее реализацию к структуре серверной части. В этом случае нам пришлось бы менять клиентский код при каждом изменении состава серверов. Прослойка веб-серверов должна скрывать эти подробности и отвечать на запросы своих клиентов как можно быстрее при любых обстоятельствах.

Возьмем, к примеру, серверный сервис, предоставляющий информацию, которая выводится на всплывающей контактной карточке при наведении указателя на имя отправителя. Учитывая остальные возможности Gmail, эта функция является второстепенной, поэтому, если соответствующий серверный сервис недоступен, веб-сервер в ответ на запрос может вернуть код временного сбоя. Клиентская сторона не должна сигнализировать об этом состоянии — она может просто перестать показывать всплывающую карточку и повторить запрос, когда пользователь опять инициирует ее появление.

Такие доводы применимы не только к прослойке веб-серверов. Большое приложение, состоящее из сотен или тысяч серверных сервисов, обязано справляться со сбоями и недоступностью подобным образом, в противном случае человеку невозможно будет понять поведение системы. По аналогии с тем, как мы разбиваем функциональность на модули, обработка внештатных ситуаций тоже должна быть разделена на предметные области.

1.5. Избегаем архитектуры вида «большой ком грязи»

На этом этапе приложение Gmail состоит из клиентской части, которая выполняется на устройстве пользователя, серверных сервисов, отвечающих за хранение данных, и веб-серверов, которые представляют собой входы в серверную часть. Последние играют важную роль не только в обеспечении отзывчивости, которую мы обсудили в предыдущем разделе, но и в разделении клиентской и серверной частей на уровне архитектуры. Наличие такой четко определенной точки входа для клиентских запросов упрощает взаимодействие между той частью приложения, которая работает на устройстве пользователя, и той, что выполняется на серверах в облаке.

Серверная часть состоит из множества сервисов, разделение и взаимодействие которых является результатом применения шаблона «простой компонент». Сам по себе этот шаблон не предоставляет сдержек и противовесов, которые не дают архитектуре превратиться в большой бардак, где почти все сервисы общаются между собой. Такой системой было бы сложно управлять; даже если применить индивидуальную обработку сбоев, предохранители и управление потоком, полностью разобраться в ней и вносить в нее изменения с уверенностью

не смог бы ни один человек. Такой сценарий имеет неформальное название *«большой ком грязи» (big ball of mud)*.

Поскольку проблема заключается в неконтролируемом взаимодействии между произвольными серверными сервисами, чтобы решить ее, следует сосредоточиться на путях коммуникации в рамках всего приложения, проектируя их определенным образом. Это называется *потоком сообщений*, о чем будет подробно рассказано в главе 15.

На рис. 1.2 показано разбиение на слишком крупные сервисы, так что это нельзя считать примером «кома грязи». Это скорее иллюстрация принципа проектирования потока сообщений в том смысле, что сервис, отвечающий за составление письма, вероятно, не должен общаться непосредственно с сервисом, формирующим всплывающую карточку. Если создание письма влечет за собой отображение карты адресата, который в нем упоминается, запрашивать эту карту должна не серверная, а клиентская часть — точно так же, как это происходит при наведении курсора на заголовок письма. Таким образом мы устраняем один потенциальный маршрут потока сообщений, упрощая общую модель взаимодействия серверных сервисов.

Еще одно преимущество тщательно продуманного потока сообщений состоит в облегчении тестирования и покрытия всех сценариев взаимодействия. Имея понятную архитектуру потока сообщений, мы можем с уверенностью сказать, с какими сервисами взаимодействует компонент и чего ожидать от него в плане пропускной способности и задержек. К данной методике можно подойти с другой стороны и использовать ее в качестве канарейки в угольной шахте: если сложно определить, в каких сценариях следует тестировать тот или иной компонент, это верный признак того, что система близка к превращению в «большой ком грязи».

1.6. Взаимодействие нереактивных компонентов

Последний важный аспект создания приложений в соответствии с реактивными принципами заключается в том, что в большинстве случаев нам придется интегрироваться с уже существующей инфраструктурой или системами, которые не обладают необходимыми характеристиками. В качестве примеров можно привести:

- драйверы устройств, которым недостает инкапсуляции и которые при сбое завершают весь процесс;
- программные интерфейсы, выполняющие свои функции синхронно, блокируя тем самым вызывающий код и не давая ему реагировать на другой ввод или истечение времени ожидания;
- системы с неограниченными входящими очередями, которые не соблюдают лимиты, налагаемые на задержки.

Для устранения большинства этих проблем используются *шаблоны управления ресурсами*, рассматриваемые в главе 14. Основная идея состоит в добавлении необходимой инкапсуляции и асинхронных границ путем взаимодействия с ресурсом в рамках отдельного реактивного компонента и применения дополнительных потоков, процессов и компьютеров по мере необходимости. Это позволяет легко интегрировать такие ресурсы в нашу архитектуру.

При взаимодействии с системой, которая не ограничивает время ожидания ответа, необходимо предусмотреть возможность сообщать о кратковременных перегрузках. Этого в какой-то степени можно достичь с помощью предохранителей, но, кроме этого, вы должны определиться с тем, какой должна быть реакция на перегрузку. В этом нам опять-таки помогут *шаблоны управления потоком*, описанные в главе 16.

В контексте приложения Gmail примером может служить гипотетическая интеграция с внешним инструментом, таким как совместный список покупок. На клиентской стороне пользователь добавляет пункты в список, извлекая нужную информацию из писем в полуавтоматическом режиме. На стороне сервера эта функция будет поддерживаться в виде сервиса, который инкапсулирует программный интерфейс внешней утилиты. Учитывая, что взаимодействие со списком покупок, скорее всего, потребует использования родной библиотеки, подверженной сбоям и способной утянуть за собой весь процесс, в котором она работает, эту задачу желательно выполнять в отдельном процессе. Затем внешний программный интерфейс в такой инкапсулированной форме интегрируется с помощью средств *межпроцессного взаимодействия* (interprocess communication, IPC) операционной системы, таких как каналы, сокеты и разделяемая память.

Можно также предположить, что реализация списка покупок использует практически неограниченную входную очередь, поэтому стоит подготовиться к возможному увеличению времени ожидания. К примеру, если на отображение элемента списка уходят минуты, это может запутать и, возможно, разочаровать пользователей. Для решения данной проблемы можно следить за списком покупок и замерять время ожидания в рамках серверного сервиса Gmail, который занимается этим взаимодействием. Если текущая задержка превышает допустимое значение, сервис должен либо ответить на запрос отказом и кодом временного сбоя, либо выполнить операцию, включив в ответ предупреждение. Затем клиентское приложение может оповестить пользователя о том или ином исходе: в одном случае оно предложит повторить попытку позже, а в другом — проинформирует о задержке.

1.7. Резюме

В данной главе вы познакомились с миром реактивных архитектур в контексте принципов, на которых базируется манифест реактивного программирования, а также рассмотрели основные вызовы, стоящие перед теми, кто разрабатывает приложения в этом стиле. Более подробный пример проектирования реактивного приложения можно найти в приложении Б. В следующей главе мы глубоко окунемся в сам манифест.