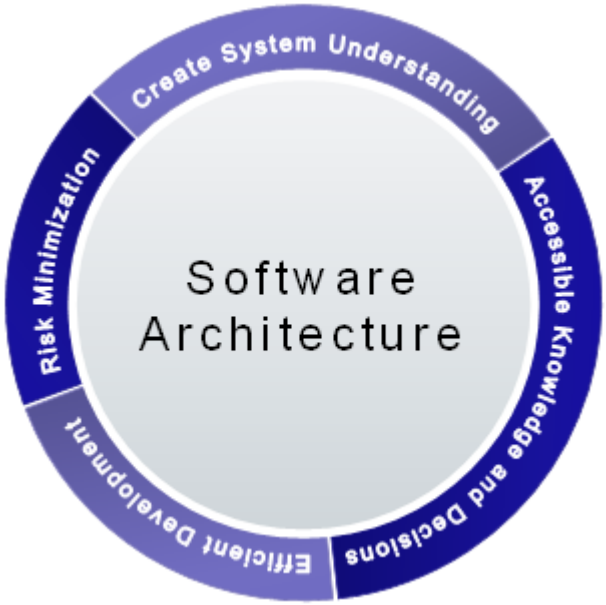
Н.А. Осипов

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОКОММУНИКАЦИОННЫХ СИСТЕМ

Учебное пособие



Санкт-Петербург

2022

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

Н.А. Осипов

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОКОММУНИКАЦИОННЫХ СИСТЕМ

УЧЕБНОЕ ПОСОБИЕ

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки (специальности) 11.04.02
Инфокоммуникационные технологии и системы связи (Магистратура) в качестве учебного пособия для реализации основных профессиональных обязательных программ высшего образования магистратуры

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург

2022

Н.А. Осипов. Архитектура программного обеспечения инфокоммуникационных систем – СПб: Университет ИТМО, 2022. – 63 с.

Рецензент(ы):
Ананченко И.В., к.т.н., доцент (квалификационная категория "доцент практики") факультета инфокоммуникационных технологий, Университет ИТМО.

В учебном пособии представлены материалы для выполнения лабораторных работ и курсового проектирования при изучении учебной дисциплины "Архитектура программного обеспечения инфокоммуникационных систем" по направлению подготовки (специальности) 11.04.02 Инфокоммуникационные технологии и системы связи (Магистратура). Учебное пособие предназначено для теоретического и практического освоения методики формирования архитектуры программного обеспечения. Для выполнения задания курсовой работы описывается процесс создания архитектуры облачного приложения как структурированного решения, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества.

УНИВЕРСИТЕТ ИТМО

Университет ИТМО – национальный исследовательский университет, ведущий вуз России в области информационных, фотонных и биохимических технологий. Альма-матер победителей международных соревнований по программированию – ICPC (единственный в мире семикратный чемпион), Google Code Jam, Facebook Hacker Cup, Яндекс.Алгоритм, Russian Code Cup, Topcoder Open и др. Приоритетные направления: IT, фотоника, робототехника, квантовые коммуникации, трансляционная медицина, Life Sciences, Art&Science, Science Communication. Входит в ТОП-100 по направлению «Автоматизация и управление» Шанхайского предметного рейтинга (ARWU) и занимает 74 место в мире в британском предметном рейтинге QS по компьютерным наукам (Computer Science and Information Systems). С 2013 по 2020 гг. – лидер Проекта 5–100.

Содержание

Введение

В данном пособии описываются лабораторные работы и курсовой проект по дисциплине "Архитектура программного обеспечения инфокоммуникационных систем" по направлению подготовки (специальности) 11.04.02 Инфокоммуникационные технологии и системы связи (Магистратура). Представленные материалы обеспечивают достижение целей дисциплины в части получения следующих результатов обучения: ПК-С3.1 (способен оценивать возможности создания, разрабатывать требования и проектировать программное обеспечение) и ПК-5 (способен разрабатывать и применять аппаратную и программную архитектуру современных сетевых устройств, конкретные методы реализации различных протоколов и сетевых сервисов, способы управления сетевыми инфокоммуникационными устройствами). После выполнения комплекса лабораторных работ и курсового проектирования обучающиеся приобретут умения и закрепят навыки в использовании принципов управления требований к программному обеспечению (ПО), формированию системы требований к программному обеспечению, построения устойчивой архитектуры ПО, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества.

Все лабораторные работы имеют общую цель – разработку архитектурного решения ПО конкретной инфокоммуникационной системы. Каждая лабораторная работа посвящена решению отдельной задачи (этапа разработки), а в итоге, после последовательного выполнения всех заданий, должно получиться архитектурное структурированное решение, отвечающее всем техническим и операционным требованиям и обеспечивающее оптимальные общие атрибуты качества, такие как производительность, безопасность и управляемость. На выполнение каждой лабораторной работы выделяется 4 часа аудиторного времени. Полностью завершить работу и оформление ее результатов обучающиеся должны в часы самостоятельной работы. Отчет представляется в формате, предусмотренном шаблоном отчета по лабораторной работе. Отчет не может быть принят и подлежит доработке в случае отсутствия необходимых разделов, отсутствия необходимого графического материала, некорректной обработки результатов, а также неполных выводов по разделам работы. Защита отчета проходит в форме доклада обучающегося по выполненной работе и ответов на вопросы преподавателя, для защиты отчета рекомендуется подготовить презентационные материалы.

После изучения лекционной части второго раздела дисциплины студенты получают задание на курсовую работу. Курсовое проектирование представляет собой особый вид учебной деятельности - самостоятельную работу студента над проектным заданием под руководством

преподавателя, главная цель которой заключается в подготовке студента к дипломному проектированию.

Цели курсового проектирования:

- систематизация, закрепление и углубление знаний, умений и навыков по разработке архитектуры программного обеспечения инфокоммуникационных систем;
- приобретение навыков проектирования и развитие творческого мышления в этом процессе;
- закрепление навыков поиска и использования справочной, нормативной и научной литературы;
- приобретение навыков оформление пояснительной записки.

Тема задания общая: «Построение архитектуры программного обеспечения облачного приложения», но каждый студент получает свою индивидуальную инфокоммуникационную систему.

Целью работы является создание архитектуры программного обеспечения системы как облачного приложения. В ходе выполнения задания студенты должны провести информационное обследование объекта, разработать необходимый набор моделей автоматизируемых процессов (рекомендуется воспользоваться заданиями лабораторных работ), сформировать техническое задание на проект. Требуется также разработать рекомендации по проектированию масштабируемых, отказоустойчивых и высокодоступных облачных приложений с учетом особенностей их применения.

Работа представляется в виде пояснительной записки по проекту, содержащей техническое описание и спецификацию. Структура пояснительной записки, а также ее оформление должно соответствовать требованиям соответствующих руководящих документов. Защита проводится в форме устного доклада с применением презентационных материалов.

Лабораторная работа 1. Проектирование архитектуры многослойного приложения

Цель работы

Изучение структуры многослойного приложения, освоение процесса моделирования предметной области для проектирования архитектуры прикладных программных систем и приобретение навыков разработки прототипа архитектуры.

Теоретические основы работы

Описание главных элементов архитектуры (задача 1)

Создание архитектуры приложения представляет собой процесс формирования структурированного решения, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества, такие как производительность, безопасность и управляемость [1, 2].

Этот процесс включает принятие ряда решений на основании широкого диапазона факторов (эти решения вы будете принимать, решая конкретные задания, указанные в практической части работы). Каждое из этих решений может иметь существенное влияние на качество и производительность приложения, а также удобство его использования.

Как и любая другая сложная структура, ПО должно строиться на прочном фундаменте. Неправильное определение ключевых сценариев, неправильное проектирование общих вопросов или неспособность выявить долгосрочные последствия основных решений могут поставить под угрозу все приложение. Современные инструменты и платформы упрощают задачу по созданию приложений, но не устраняют необходимости в тщательном их проектировании на основании конкретных сценариев и требований. Неправильно выработанная архитектура обусловливает нестабильность ПО, невозможность поддерживать существующие или будущие бизнес-требования, сложности при развертывании или управлении в среде производственной эксплуатации [2].

Проектирование систем должно осуществляться с учетом потребностей пользователя, системы (ИТ-инфраструктуры) и бизнес-целей. Для каждой из этих составляющих определяются ключевые сценарии и выделяются важные параметры качества (например, надежность или масштабируемость), а также основные области удовлетворенности и неудовлетворенности. По возможности необходимо выработать и учесть показатели успешности в каждой из этих областей [1]. В практической части работы этому посвящена задача №1. Основная рекомендация при решении задачи №1 – определять компромиссы и находить баланс между конкурирующими требованиями этих трех

областей. Например, пользовательский интерфейс решения очень часто определяется бизнес-целями и ИТ-инфраструктурой в целом, и изменения одного из этих компонентов могут существенно влиять на результирующие механизмы взаимодействия с пользователем. Аналогично, изменения в требованиях по взаимодействию с пользователем могут оказывать сильное воздействие на требования к бизнес-области и ИТ-инфраструктуре.

Таким образом, архитектор должен учитывать общий эффект от принимаемых проектных решений, обязательно присутствующие компромиссы между атрибутами качества и компромиссы, необходимые для выполнения пользовательских, системных и бизнес-требований.

Проектирование прототипа архитектуры (задача 2)

При проектировании архитектуры приложения следует учитывать, что архитектура должна объединять бизнес-требования и технические требования через сценарии – варианты использования (Use Case) с последующим нахождением возможных их реализаций в алгоритмах ПО [1].

Прототип архитектуры должен основываться на требованиях (полученных из предыдущего пункта), оказывающих влияние на структуру приложения.

Следует учитывать признаки хорошей архитектуры, которая снижает бизнес-риски, связанные с созданием технического решения, и обладает значительной гибкостью, чтобы справляться с естественным развитием технологий, как в области оборудования и ПО, так и пользовательских сценариев и требований.

При проектировании архитектуры руководствуйтесь следующими основными принципами [1]:

- Создавайте с готовностью к изменениям – продумайте, как со временем может понадобиться изменить приложение, чтобы оно отвечало вновь возникающим требованиям и задачам, и предусмотрите необходимую гибкость.
- Создавайте модели для анализа и сокращения рисков – используйте средства проектирования, системы моделирования, такие как Унифицированный язык моделирования (Unified Modeling Language, UML), и средства визуализации, когда необходимо выявить требования, принять архитектурные и проектные решения и проанализировать их последствия. Тем не менее, не создавайте слишком формализованную модель, она может ограничить возможности для выполнения итераций и адаптации дизайна.
- Используйте модели и визуализации как средства общения при совместной работе. Используйте модели, представления и другие способы визуализации архитектуры для эффективного обмена

информацией и связи со всеми заинтересованными сторонами, а также для обеспечения быстрого оповещение об изменениях в дизайне.

- Рассмотрите возможность использования инкрементного и итеративного подхода при работе над архитектурой. Начинайте с базовой архитектуры, воссоздавая полную картину, и затем прорабатывайте возможные варианты в ходе итеративного тестирования и доработки архитектуры.

Таким образом, при создании прототипа (задача №2) необходимо помнить, что архитектура должна:

- раскрывать структуру системы, но скрывать детали реализации;
- реализовывать все варианты использования и сценарии;
- по возможности отвечать всем требованиям различных заинтересованных сторон;
- выполнять требования, как по функциональности, так и по качеству.

Применение компонентов на всех слоях разрабатываемого приложения (задача 3)

Архитектуру ПО можно описать как организацию или структуру системы, где система представляет набор компонентов, выполняющих определенную функцию или набор функций. В этом смысле основная задача архитектуры состоит в организации компонентов с целью обеспечения определенной функциональности. Такую организацию функциональности часто называют группировкой компонентов по «функциональным областям» [1].

На рис. 1 представлена типовая архитектура приложения, компоненты которого сгруппированы по функциональным областям, но следует учитывать, что функциональные области используются не только для группировки компонентов, а могут быть применимы для реализации взаимодействия и организации совместной работы компонентов.

При решении задачи №3 (проектирование архитектуры как взаимодействие компонентов) следует учитывать рекомендации по проектированию различных функциональных областях [1]:

- Разделение функций. Разделите приложение на отдельные компоненты с учетом минимального перекрытия функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (high cohesion) и слабую связанность (low coupling). Неверное разграничение функциональности может привести к высокой связанности и сложностям взаимодействия, даже несмотря на слабое перекрытие функциональности отдельных компонентов. Основное преимущество такого подхода – независимая оптимизация

функциональных возможностей, и кроме того, сбой одной из функций не приведет к сбою остальных, поскольку они могут выполняться независимо друг от друга. Такой подход также упрощает понимание и проектирование приложения и облегчает управление сложными взаимосвязанными системами.

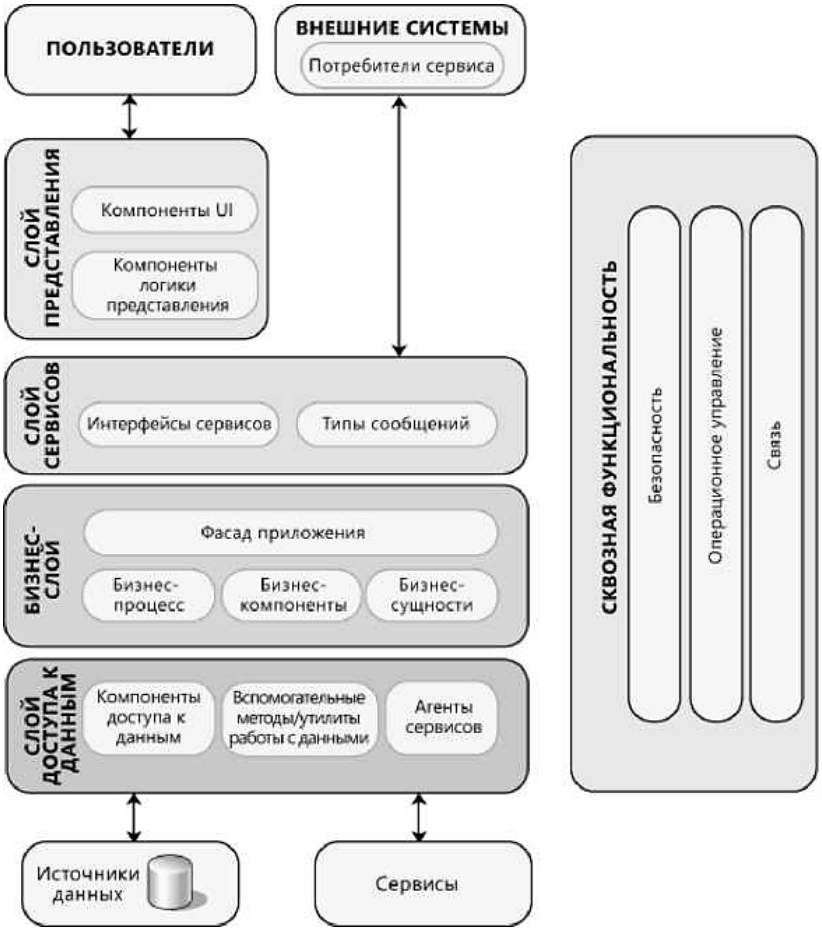


Рис. 1 Типовая архитектура приложения

- Принцип единственности ответственности. Каждый отдельно взятый компонент или модуль должен отвечать только за одно конкретное свойство/функцию или совокупность связанных функций.
- Принцип минимального знания (Закон Деметера (Law of Demeter, LoD)). Компоненту или объекту не должны быть известны внутренние детали других компонентов или объектов.
- Не повторяйтесь (Don't repeat yourself, DRY). Намерение должно быть обозначено только один раз. В применении к проектированию приложения это означает, что определенная функциональность должна быть реализована только в одном компоненте и не должна дублироваться ни в одном другом компоненте.
- Минимизируйте проектирование наперед. Проектируйте только то, что необходимо. Если требования к приложению четко не определены, или существует вероятность изменения дизайна со

временем, старайтесь не тратить много сил на проектирование раньше времени. Этот принцип называют YAGNI («You ain't gonna need it»).

- Определяйте четкий контракт для компонентов. Компоненты, модули и функции должны определять контракт или спецификацию интерфейса, четко оговаривающую их использование и поведение. Контракт должен описывать, как другие компоненты могут выполнять доступ к внутренней функциональности компонента, модуля или функции, и поведение этой функциональности с точки зрения предварительных условий, постусловий, побочных эффектов, исключений, рабочих характеристик и других факторов.
- Максимально изолируйте сквозную функциональность от бизнес-логики приложения. Сквозная функциональность включает в себя аспекты безопасности, обмена информацией или управляемости, такие как протоколирование и инструментирование. Смешение кода, реализующего эти функции, с бизнес-логикой может привести к созданию дизайна, который будет сложно расширять и обслуживать. Внесение изменений в сквозную функциональность потребует переработки всего кода бизнес-логики.

Таким образом, важная цель архитектора ПО при проектировании приложения состоит в максимальном упрощении дизайна через его разбиение на функциональные области. Например, к разным функциональным областям можно отнести пользовательский интерфейс (user interface, UI), выполнение бизнес-процессов и доступ к данным. Компоненты в каждой из этих областей должны реализовывать конкретную функциональность и не должны смешивать в себе код разных функциональных областей. Так, в компонентах UI не должно быть кода прямого доступа к источнику данных – для извлечения данных в них должны использоваться либо бизнес-компоненты, либо компоненты доступа к данным.

Практические задания

Задача 1. Сформируйте описание главных элементов архитектуры (пользователь, система, бизнес-цели) и для каждого элемента сформулируйте ключевые сценарии и выберите показатели качества.

Дайте развернутые ответы на следующие вопросы:

1. Как пользователь будет использовать приложение?
2. Как приложение будет развертываться и обслуживаться при эксплуатации?
3. Какие требования по атрибутам качества, таким как безопасность, производительность, возможность параллельной обработки, интернационализация и конфигурация, выдвигаются к приложению?

4. Как спроектировать приложение, чтобы оно оставалось гибким и удобным в обслуживании в течение долгого времени?
5. Какие существуют основные архитектурные направления, которые могут оказывать влияние на приложение сейчас или после его развертывания?

Задача 2. Разработайте прототип архитектуры. Предложите вариант разбиения приложения на слои.

При проектировании архитектуры необходимо ответить на следующие вопросы:

1. Какие части архитектуры являются фундаментальными, изменение которых в случае неверной реализации представляет наибольшие риски?
2. Какие части архитектуры, вероятнее всего, подвергнутся изменениям, а также проектирование каких частей можно отложить?
3. Основные допущения и как они будут проверяться?
4. Какие условия могут привести к реструктуризации дизайна?
5. Существуют ли явные решения о зависимостях между слоями и о потоках данных между ними?

Задача 3. Разработайте предварительные предложения по применению компонентов на всех слоях разрабатываемого приложения. Перечислите, какие задачи эти компоненты должны решать в целях достижения требуемой функциональности.

При проектировании компонентов необходимо ответить на следующие вопросы:

1. Какие важные (на данном этапе проектирования) функции (свойства) системы можно поручить отдельным компонентам?
2. Можно ли сейчас отнести компоненты к определенному типу?
3. Как объединить отдельные компоненты по функциональным областям?
4. Дублируется ли функциональность в различных компонентах?
5. Какие индивидуальные эксплуатационные требования компонентов можно выделить на данном этапе?
6. Располагает ли каждый метод, вызываемый компонентом, достаточными сведениями о том, как обрабатывать поступающие запросы, а в случае необходимости как перенаправлять их к соответствующим подкомпонентам или другим компонентам?
7. Какие предполагаются сценарии развертывания приложения, будут ли все компоненты выполняться в рамках одного процесса, или необходимо обеспечить поддержку связи между компонентами, разворачиваемых в различных процессах?

8. Как компоненты будут реализовывать сквозную функциональность?

Отчетность по работе

Отчет должен содержать:

1. Титульный лист.
2. Цель и задачи лабораторной работы.
3. Результаты выполнения практической части.
4. Диаграммы и другие графические материалы.
5. Выводы по работе.

Рекомендации по формулировке выводов: например, …был проведен первичный анализ области применения системы …. После определения основных сущностей системы была принято решение в пользу архитектуры …. Проведенный анализ позволил в общих чертах … . Дальнейшее проектирование должно быть направлено на …..

Контрольные вопросы

1. Что такое архитектура программного обеспечения?
2. Перечислите характерные черты архитектуры программного обеспечения.
3. Какие вопросы из нижеперечисленных должен задать архитектор веб-приложения при разработке архитектуры:
    a. какой нужен макет страниц?
    b. сколько одновременно работающих пользователей должно поддерживать приложение?
    c. как необходимо организовать деревья навигации?
    d. существуют ли в среде хостер-провайдера ограничения на применяемые технологии?
4. Что представляет собой сквозная функциональность? Укажите потенциальные проблемы от смешивания кода, реализующего эти функции, с бизнес-логикой?

Лабораторная работа 2. Реализация модели архитектуры приложения в виде артефакта RUP

Цель работы

Изучение типов приложений многослойного приложения, освоение процесса моделирования архитектуры прикладных программных систем и приобретение навыков разработки модели архитектуры в рамках унифицированного процесса разработки (RUP).

Теоретические основы работы

Требуется на основе полученных результатов первой работы создать модель архитектуры приложения – организовать компоненты с целью обеспечения определенной (заданной) функциональности, т.е. разработать группировку компонентов по «функциональным областям».

В этой работе вы примите основные решения, которые помогут учесть важные факторы при итеративной разработке проекта архитектуры.

Определение типа приложения (задача 1)

Ключевым моментом процесса проектирования приложения является выбор соответствующего типа приложения. Этот выбор определяется конкретными требованиями и ограничениями среды, в частности, требование поддержки множества типов клиентов и возможность использования более одного базового архетипа.

Как правило, при проектировании архитектуры рассматривают следующие основные типы приложений [1]:

- Приложения для мобильных устройств.
- Насыщенные клиентские приложения для выполнения преимущественно на клиентских ПК.
- Насыщенные клиентские приложения для развертывания из Интернета с поддержкой насыщенных UI и мультимедийных сценариев.
- Сервисы, разработанные для обеспечения связи между слабо связанными компонентами.
- Веб-приложения для выполнения преимущественно на сервере в сценариях с постоянным подключением.
- Приложения и сервисы, размещаемые в центрах обработки данных (ЦОД) и в облаке.
- Офисные бизнес-приложения (Office Business Applications, OBAs), интегрирующие технологии Microsoft Office и Microsoft Server.
- Бизнес-приложения SharePoint (SharePoint Line of Business, LOB), обеспечивающие доступ к бизнес-данным и функциональным возможностям через портал.

Реализация сквозной функциональности (задача 2)

Сквозная функциональность представляет область дизайна, не связанную с конкретным функционалом приложения. При проектировании архитектуры следует принять решение о возможности распределения функционала между слоями для следующих аспектов [1, 2]:

- Механизм протоколирования. Обеспечивайте управление и мониторинг всех критически важных для бизнес-логики и системы событий. Протоколируйте достаточное количество сведений для

воссоздания событий в системе без включения конфиденциальных данных. Рекомендуется реализовать возможность каждому слою вести журнал в общем хранилище либо в разных хранилищах, но таким образом, чтобы результаты могли быть сопоставлены впоследствии.

- Аутентификация. Определитесь с тем, как будет проходить аутентификация пользователей и передача аутентифицированных удостоверений между слоями.
- Авторизация. На каждом уровне и между границами доверия обеспечьте соответствующую авторизацию с необходимой детализацией.
- Управление исключениями. Перехватывайте исключения на функциональных, логических и физических границах и избегайте раскрытия конфиденциальных сведений конечным пользователям. Инфраструктура управления исключениями должна функционировать в каждом слое и между уровнями, если исключения распространяются в рамках системы.
- Связь. Выберите подход к реализации связей, используемый для обеспечения обмена информацией между слоями, соответствующие протоколы, сведите до минимума количество вызовов по сети и защитите передачу конфиденциальных данных по сети.
- Кэширование. Определите, что должно кэшироваться и где для улучшения производительности и сокращения времени отклика приложения. Общая инфраструктура кэширования должна позволять кэшировать данные в слое представления, бизнес-слое и слое доступа к данным.

Выбор типа архитектуры (задача 3)

Помимо типа разрабатываемого приложения, важным фактором при выборе архитектуры приложения является выбор архитектурного стиля – архитектурного шаблона – набора принципов как высокоуровневой схемы, обеспечивающей абстрактную инфраструктуру для семейства систем. Ожидается, что архитектурный стиль улучшает секционирование и способствует повторному использованию дизайна благодаря обеспечению решений часто встречающихся проблем. Архитектурные стили и шаблоны можно рассматривать как набор принципов, формирующих приложение. К ним относят такие шаблоны, как клиент/сервер, многоуровневая архитектура, компонентная архитектура, архитектура, основанная на шине сообщений, и сервисно-ориентированная архитектура (service-oriented architecture, SOA).

При выборе архитектурного шаблона необходимо рассмотреть суть его построения, основные принципы и преимущества, которые помогут сформировать правильную архитектуру.

Далее приводится список типовых архитектурных стилей и дается краткое описание каждого из них, а также рекомендации по выбору соответствующих стилей для конкретного приложения [1].

- Клиент/сервер. Система разделяется на два приложения, где клиент выполняет запросы к серверу. Во многих случаях в роли сервера выступает база данных, а логика приложения представлена процедурами хранения.
- Компонентная архитектура. Дизайн приложения разлагается на функциональные или логические компоненты с возможностью повторного использования, предоставляющие тщательно проработанные интерфейсы связи.
- Дизайн на основе предметной области. Представляет объектно-ориентированный архитектурный стиль, ориентированный на моделирование сферы деловой активности и определяющий бизнес-объекты на основании сущностей этой сферы.
- Многослойная архитектура. Функциональные области приложения разделяются на многослойные группы.
- Шина сообщений. Архитектурный стиль, предписывающий использование программной системы, которая может принимать и отправлять сообщения по одному или более каналам связи, так что приложения получают возможность взаимодействовать, не располагая конкретными сведениями друг о друге.
- N-уровневая / 3-уровневая. Функциональность выделяется в отдельные сегменты – уровни, во многом аналогично многослойному стилю, но в данном случае сегменты физически располагаются на разных компьютерах.
- Объектно-ориентированная. Парадигма проектирования, основанная на распределении ответственности приложения или системы между отдельными многократно используемыми и самостоятельными объектами, содержащими данные и поведение.
- Сервисно-оринетрированная архитектура (SOA). Описывает приложения, предоставляющие и потребляющие функциональность в виде сервисов с помощью контрактов и сообщений.

Важно понимать, что стили оказывают определенное влияние на архитектуру и описывают разные аспекты приложений, например, некоторые архитектурные стили описывают схемы развертывания, некоторые – вопросы структуры и дизайна, а другие – аспекты связи. Таким образом, типовое приложение, как правило, использует сочетание нескольких архитектурных стилей, образующих полную систему. Например, может существовать SOA-дизайн, состоящий из сервисов, при разработке которых использовалась многослойная архитектура и объектно-ориентированный архитектурный стиль.

Сочетание архитектурных стилей также полезно при построении Интернет веб-приложений, где можно достичь эффективного разделения функциональности за счет применения многослойного архитектурного стиля. Таким образом можно отделить логику представления от бизнес-логики и логики доступа к данным. Требования безопасности организации могут обусловливать либо трех-уровневое развертывание приложения, либо развертывание с более чем тремя уровнями. Уровень представления может развертываться в пограничной сети, располагающейся между внутренней сетью организации и внешней сетью. В качестве модели взаимодействия на уровне представления может применяться шаблон представления с отделением (разновидность многослойного стиля), такая как Model-View-Controller (MVC). Также можно выбрать архитектурный стиль SOA и реализовать связь между веб-сервером и сервером приложений посредством обмена сообщениями.

Создавая настольное приложение, можно реализовать клиентское приложение, которое будет отправлять запросы к программе на сервере. В этом случае развертывание клиента и сервера можно выполнить с помощью архитектурного стиля клиент/сервер и использовать компонентную архитектуру для дальнейшего разложения дизайна на независимые компоненты, предоставляющие соответствующие интерфейсы. Применение объектно-ориентированного подхода к этим компонентам повысит возможности повторного использования и тестирования.

Применение компонентов при построении архитектуры (задача 4)

Компонентная архитектура описывает подход к проектированию и разработке систем, заключающийся в разложении дизайна на отдельные функциональные или логические компоненты, предоставляющие четко определенные интерфейсы, содержащие методы, события и свойства. Термин “компонент” часто используется и в более общем смысле как составная часть, элемент или составляющая.

В этом случае обеспечивается более высокий уровень абстракции, чем при объектно-ориентированной разработке, и не происходит концентрации внимания на таких вопросах, как протоколы связи или общее состояние.

Основной принцип компонентного стиля заключается в применении компонентов, обладающих следующими качествами [1]:

- Пригодность для повторного использования. Компоненты проектируются с обеспечением возможности их повторного использования в разных сценариях различных приложений.
- Замещаемость. Компоненты должны заменяться другими подобными компонентами.
- Независимость от контекста. Компоненты проектируются для работы в разных средах и контекстах. Специальные сведения, такие

как данные о состоянии, должны не включаться или извлекаться компонентом, а передаваться в него.

- Расширяемость. Компонент может расширять существующие компоненты для обеспечения нового поведения.
- Инкапсуляция. Компоненты предоставляют интерфейсы, позволяющие вызывающей стороне использовать их функциональность, не раскрывая при этом детали внутренних процессов либо внутренние переменные или состояние.
- Независимость. Компоненты проектируются с минимальными зависимостями от других компонентов. Таким образом, компоненты могут быть развернуты в любой подходящей среде без влияния на другие компоненты или системы.

Как правило, в приложениях используются следующие компоненты:

– компоненты пользовательского интерфейса (виджеты, элементы управления), например, кнопки, списки, контейнеры.
– вспомогательные или служебные компоненты, предоставляющие определенный набор функций, используемых в других компонентах, например, валидаторы, работа с базой данных.
– ресурсоемкие компоненты, доступ к которым осуществляется нечасто и активация которых выполняется «точно вовремя» (just-in-time, JIT) (обычно используется в сценариях с удаленными или распределенными компонентами);
– компоненты с очередью, вызовы методов которых могут выполняться асинхронно за счет применения очереди сообщений, для хранения и пересылки.

Процесс создания компонентов зависит от механизмов платформы, обеспечивающей среду выполнения и зачастую эти механизмы называют просто компонентной архитектурой.

Примерами такой архитектуры могут служить объектная модель программных компонентов (component object model, COM) и объектная модель распределенных программных компонентов (distributed component object model, DCOM) в Windows, общая архитектура брокера объектных запросов (Common Object Request Broker Architecture, CORBA) и серверные компоненты Java (Enterprise JavaBeans, EJB) на других платформах. Используемая компонентная архитектура описывает механизмы размещения компонентов и их интерфейсов, передачи сообщений или команд между компонентами и, в некоторых случаях, сохранения состояния.

Microsoft .NET Framework также предоставляет поддержку для построения приложений с использованием компонентного подхода, например, это бизнес-компоненты и компоненты данных, которые обычно являются классами, скомпилированными в сборки .NET Framework. Они

выполняются под управлением среды выполнения .NET Framework. Каждая сборка может включать более одного подобного компонента.

При проектировании архитектуры следует учитывать преимущества компонентного архитектурного стиля [1]:

- Простота развертывания. Существующие версии компонентов могут заменяться новыми совместимыми версиями, не оказывая влияния на другие компоненты или систему в целом.
- Меньшая стоимость. Использование компонентов сторонних производителей позволяет распределять затраты на разработку и обслуживание.
- Простота разработки. Для обеспечения заданной функциональности компоненты реализуют широко известные интерфейсы, что позволяет вести разработку без влияния на другие части системы.
- Возможность повторного использования. Применение многократно используемых компонентов означает возможность распределения затрат на разработку и обслуживание между несколькими приложениями или системами.
- Упрощение с технической точки зрения. Компоненты упрощают систему через использование контейнера компонентов и его сервисов. В качестве примеров сервисов, предоставляемых контейнером, можно привести активацию компонентов, управление жизненным циклом, организацию очереди вызовов методов, обработку событий и транзакции.

Для управления зависимостями между компонентами, поддержки слабого связывания и повторного использования могут использоваться шаблоны проектирования, такие как Dependency Injection (Внедрение зависимостей) или Service Locator (Локатор сервиса). Такие шаблоны часто применяются при создании составных приложений, сочетающих и повторно использующих компоненты во многих приложениях.

Рекомендуется рассмотреть возможность применения компонентой архитектуры, если вы уже располагаете подходящими компонентами или можете получить их от сторонних производителей, предполагается, что создаваемое приложение будет преимущественно выполнять процедурные функции, возможно, с небольшим количеством вводимых данных или вообще без таковых, или вы хотите иметь возможность сочетать компоненты, написанные на разных языках программирования.

Создании прототипа архитектуры (задача 5)

В этой части руководства рассматривается итеративная методика для создания прототипа архитектуры. Эта методика поможет свести воедино ключевые решения, обсуждаемые при проектировании, включая решения по параметрам качества, архитектурным стилям, типам приложений, технологиям и сценариям развертывания. Данная методика предполагает

создание архитектуры в ходе процесса, состоящего из серий по пять основных шагов, разбитых в свою очередь на отдельные элементы.

Итеративный процесс помогает выработать возможные варианты решений, которые дорабатываются в ходе итераций и, в конечном счете, обеспечивают создание дизайна архитектуры, наиболее соответствующей разрабатываемому приложению. Модель архитектуры может многократно пересматриваться в ходе жизненного цикла проекта, когда происходит дальнейшая доработка архитектуры, дополнения ее новыми аспектами, выявленными в последующий период сбора сведений, создания прототипов и фактической разработки.

Исходные данные проектирования помогают формализовать требования и ограничения, которые должна реализовать создаваемая архитектура.

Исходными данными являются [1, 2]:

- варианты использования и сценарии поведения пользователя;
- функциональные и нефункциональные требования;
- технологические требования;
- целевая среда развертывания;
- различного рода ограничения.

На первом шаге создается список возможных архитектурных решений, значимых с точки зрения вариантов использования, аспектов архитектуры, требующих специального внимания и которые удовлетворяют требованиям и ограничениям, выявленным в процессе проектирования.

Общей техникой постепенной доработки модели архитектуры до тех пор, пока она не будет удовлетворять всем требованиям и ограничениям, является итеративная методика, включающая пять основных этапов, как показано на рис. 2:

1. Определение целей архитектуры. Наличие четких целей поможет сосредоточиться на архитектуре и правильном выборе проблем. Точно обозначенные цели помогают определить границы каждой фазы: момент, когда завершена текущая фаза и все готово для перехода к следующей.

2. Основные сценарии. Сосредоточьтесь на том, что имеет первостепенное значение, и проверяйте возможные варианты архитектур на соответствие этим сценариям.

3. Общее представление приложения. Определите тип приложения, архитектуру развертывания, архитектурные стили и технологии, чтобы обеспечить соответствие вашего дизайна реальным условиям, в которых будет функционировать создаваемое приложение.

4. Потенциальные проблемы. Выявите основные проблемные области на основании параметров качества и потребности в сквозной функциональности.

5. Варианты решений. В каждой итерации должен быть создан «пилотный» вариант или прототип архитектуры, являющийся развитием и доработкой решения. Прежде чем переходить к следующей итерации, необходимо убедиться в соответствии этого прототипа основным сценариям, проблемам и ограничениям развертывания.

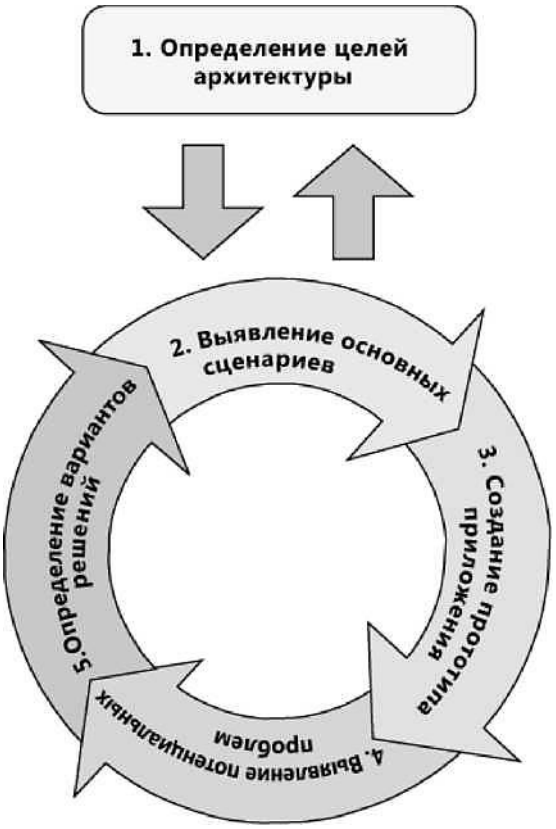


Рис. 2. Основные этапы итеративного процесса проектирования архитектуры

Такой процесс создания архитектуры предполагает итеративный и инкрементный подход. Сначала создается возможный вариант архитектуры – обобщенный дизайн, который может тестироваться по основным сценариям, требованиям, известным ограничениям и параметрам качества.

В ходе доработки варианта архитектуры выявляются дополнительные детали и сведения о дизайне, результатом чего становится расширение основных сценариев, корректировка общего представления приложения и подхода к решению проблем.

При итеративном походе к архитектуре часто есть соблазн выполнять итерации в горизонтальном направлении, в рамках отдельных слоев

приложения, а не в вертикальном направлении, что заставляет думать о функциональности, выходящей за рамки слоев и составляющей отдельную возможность (вариант использования), значимую для пользователей. При выполнении итераций в горизонтальной плоскости есть угроза реализации большого числа функций до того, как пользователи смогут их проверить.

Определив основные проблемы, можно приступать к созданию исходной базовой архитектуры и ее детализации для получения возможного варианта архитектуры.

В ходе этого процесса можно использовать пилотные архитектуры для более подробного рассмотрения определенных областей дизайна или для проверки новых идей.

Затем выполняется проверка нового варианта архитектуры на соответствие основным сценариям и заданным требованиям и вновь повторяется итеративный цикл по доработке дизайна.

Важно, особенно если проектирование и разработка ведутся по гибкому процессу, чтобы итерация включала как проектирование архитектуры, так и по разработку реализации. Это поможет избежать масштабного проектирования наперед.

Базовая архитектура описывает существующую систему, то, как она выглядит сегодня. Для нового проекта исходная базовая архитектура — это первое высокоуровневое представление архитектуры, на основании которого будут создаваться возможные варианты архитектуры.

Возможный вариант архитектуры включает:

- тип приложения;
- архитектуру развертывания;
- архитектурный стиль;
- выбранные технологии;
- параметры качества;
- сквозную функциональность.

Будьте готовы к тому, что архитектуру придется изменять несколько раз, использовать несколько итераций, возможных вариантов и множество пилотных архитектур. Если возможный вариант архитектуры является улучшением, он может стать базой для создания и тестирования новых возможных вариантов.

Итеративный и инкрементный подход позволяет избавиться от начальных крупных рисков, итеративно формировать архитектуру и через тестирование подтверждать, что каждая новая базовая архитектура является улучшением предыдущей.

Рекомендация: не стремитесь создать архитектуру за одну итерацию – каждая итерация должна раскрывать дополнительные детали, но не увязайте в деталях, сосредоточьтесь на основных этапах и создавайте инфраструктуру, на которой может быть основана ваша архитектура.

Представление прототипа архитектуры (задача 6)

Графическое представление разрабатываемой архитектуры имеет главную цель – показать основные ограничения и принятые решения для того, чтобы обозначить границы модели и начать разработку прототипа. С другой стороны, если вы не можете наглядно представить архитектуру, значит, вы не полностью понимаете ее. Если вы смогли изобразить четкую и краткую диаграмму, она будет понятной остальным, и будет намного проще объяснять детали. Например, так может выглядеть (см. рис. 3) графическое представление дизайна веб-приложения в первом приближении с указанием протоколов и методов аутентификации, которые предполагается использовать.

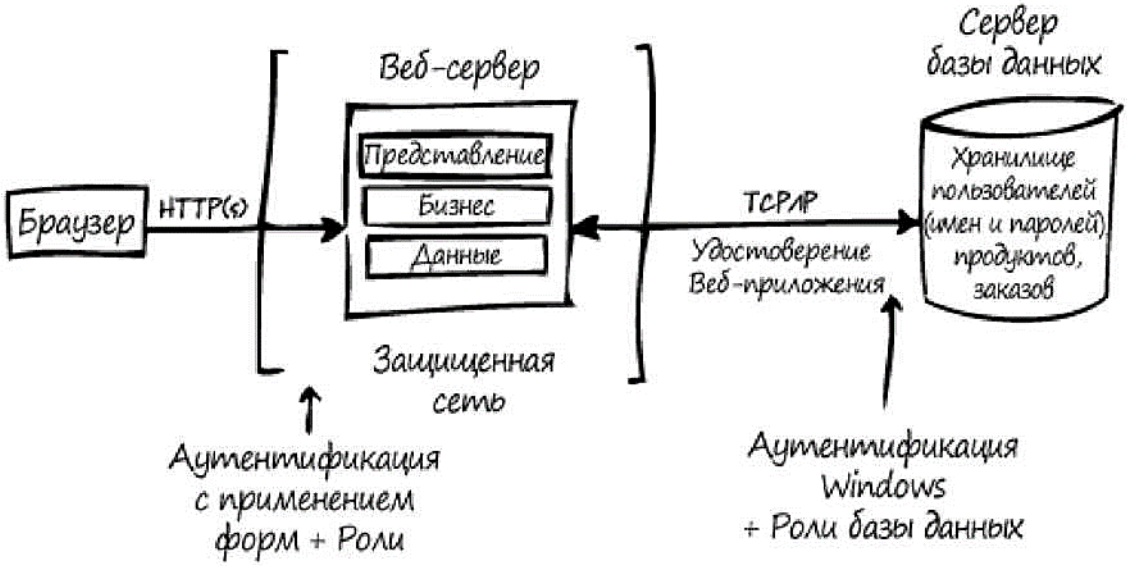


Рис. 3 Пример графического представления дизайна веб-приложения в первом приближении с указанием протоколов и методов аутентификации

Практические задания

Задача 1. Сформулируйте обоснованные предположения о типе разрабатываемого приложения.

Задача 2. Сформулируйте обоснованные предположения об элементах сквозной функциональности.

Задача 3. Опишите возможности применения архитектурных стилей (парадигм), для разрабатываемого приложения ([1], глава 3).

Задача 4. Обоснуйте возможность реализации какой-либо части приложения в виде компонента и его применения в системе.

Задача 5. Создайте модель прототипа архитектуры, раскрывающее общее представление о том, как будет выглядеть готовое приложение ([1], главы 4, 5).

Задача 6. Представьте графически (в любом редакторе) разрабатываемую архитектуру в том виде, в котором вы понимаете ее на текущем этапе.

Отчетность по работе

Отчет должен содержать:

1. Титульный лист.
2. Цель и задачи лабораторной работы.
3. Результаты выполнения практической части.
4. Диаграммы и другие графические материалы.
5. Выводы по работе.

Контрольные вопросы

1. Раскройте понятие архитектурного стиля.
2. Перечислите основные показатели качества, характерные для начальных этапов проектирования архитектуры программного обеспечения.
3. Укажите основные элементы сквозной функциональности.
4. Перечислите основные архитектурные шаблоны и их основные характеристики.
5. Приведите примеры простейшей формы системы клиент/сервер.
6. Назовите основные преимущества архитектурного стиля клиент/сервер.
7. Укажите условия, при которых целесообразно применять модель архитектуры клиент/сервер.
8. В чем суть компонентной архитектуры?
9. В чем суть многослойной (Layered) архитектуры?
10. В чем разница между слоем (layer) и уровнем (tier)?

Лабораторная работа 3. Проектирование интерфейса пользователя

Цель работы

Изучение основных рекомендаций по проектированию слоя представления, освоение процесса проектирования интерфейса пользователя (UI) в типовой архитектуре многослойного приложения и приобретение навыков разработки компонентов слоя представления многослойной архитектуры.

Теоретические основы работы

Стиль взаимодействия с пользователем и пользовательский интерфейс определяет эффективность получения от приложения требуемых действий для пользователя [3]. Видимая производительность имеет намного большее значение, чем фактическая, поэтому решающее значение имеют управление ожиданиями и знание типовых схем взаимодействия с пользователем. Например, возможно, пользователи не будут против немного дольше подождать загрузки страницы, если получают обратную связь со сведениями о предположительных сроках загрузки и это ожидание не мешает продолжению их работы. С другой стороны, есть ситуации, в которых даже очень короткая задержка для некоторых действий UI может создать ощущение, что приложение не отвечает.

Рекомендуется на первых этапах процесса разработки включать вопросы, связанные с исследованием удобства использования, опросы и интервью, чтобы понять, что пользователи хотят и ожидают от приложения, и на основании этих сведений проектировать эффективный UI [3].

Модель слоя представления (задача 1)

Слой представления содержит компоненты, реализующие и отображающие пользовательский интерфейс, а также управляющие взаимодействием с пользователем. Этот слой, кроме компонентов, организовывающих взаимодействие с пользователем, включает элементы управления для ввода данных пользователем и их отображения.

На рис. 4 показано место слоя представления в общей архитектуре приложения.

Слой представления обычно включает следующие компоненты [1]:

- Компоненты пользовательского интерфейса. Это визуальные элементы приложения, используемые для отображения данных пользователю и приема пользовательского ввода.
- Компоненты логики представления. Логика представления реализуется кодом приложения, определяющим поведение и структуру приложения и не зависящим от конкретной реализации пользовательского интерфейса.

При реализации шаблона Separated Presentation могут использоваться следующие компоненты логики представления: презентатор (Presenter), модель презентации (Presentation Model) и модель представления (View Model).

Слой представления также может включать компоненты модели слоя представления (Presentation Layer Model), которые инкапсулируют данные бизнес-слоя, или компоненты сущности представления (Presentation Entity), которые инкапсулируют бизнес-логику и данные в форме, удобной для использования слоем представления.

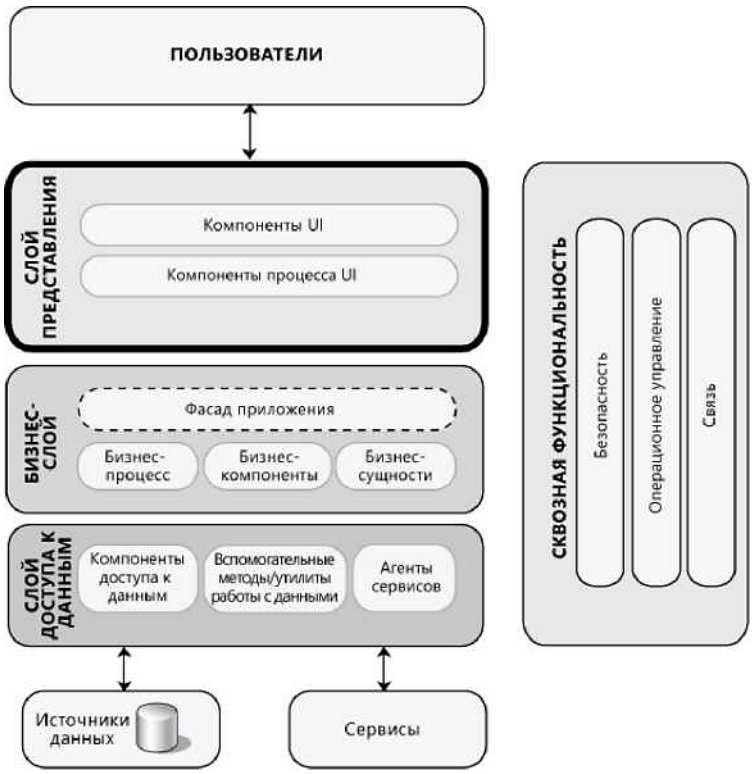


Рис. 4. Слой представления в типовом приложении

Для того чтобы создаваемая реализация UI отвечала требованиям приложения, рекомендуется следовать лучшим практикам и руководствоваться следующими принципами [1, 3]:

- Выбирайте тип приложения, соответствующий целям и решаемым задачам. От выбранного типа приложения будут существенно зависеть доступные варианты реализации слоя представления. Определитесь, будете ли вы реализовывать насыщенный клиент, веб-клиент или насыщенное Интернет-приложение (rich Internet application, RIA). Решение должно приниматься на основании требований, предъявляемых к приложению, и ограничений, накладываемых организацией или средой.
- Выбирайте соответствующую технологию UI. Разные типы приложений обеспечивают разные наборы технологий для разработки слоя представления. Каждый тип технологии обладает индивидуальными преимуществами, которые определяют возможность создания соответствующего слоя представления.
- Используйте соответствующие шаблоны. Шаблоны слоя представления предлагают проверенные решения обычных проблем, возникающих при проектировании слоя представления. Помните, что не все шаблоны применимы в равной степени ко всем архетипам, но общий шаблон Separated Presentation, в котором аспекты, касающиеся представления, отделены от базовой логики

приложения, подходит для всех типов приложений. Специальные шаблоны, такие как MVC, MVP и Supervising Presenter, обычно используются в слое представления насыщенных клиентских приложений и RIA. Разновидности шаблонов MVC и MVP могут применяться в веб-приложениях.

- Разделяйте функциональные области. Используйте специальные компоненты UI для формирования визуального представления, отображения и взаимодействия с пользователем. В сложных случаях, или если хотите обеспечить возможность модульного тестирования, обратите внимание на специальные компоненты логики представления для управления обработкой взаимодействия с пользователем.
- Учитывайте рекомендации по проектированию пользовательского интерфейса, включая такие аспекты, как удобство и простота доступа, локализация и удобство использования. Для выбранных типа клиента и технологий ознакомьтесь с установленными рекомендациями по интерактивности UI, удобству использования, совместимости с системой, соответствию предъявляемым требованиям, а также с шаблонами проектирования UI, и примените те из них, которые соответствуют дизайну и требованиям создаваемого приложения.
- Придерживайтесь принципов ориентированного на пользователя проектирования. До того, как приступать к проектированию слоя представления, поймите своего потребителя. Используйте опросы, исследования удобства использования и интервью для выбора варианта пользовательского интерфейса, наиболее соответствующего требованиям заказчика.
- Обработка длительных запросов должна реализовываться с учетом времени отклика пользователя, а также удобства обслуживания и тестируемости кода. При проектировании обработки запросов руководствуйтесь следующими рекомендациями:
  - используйте асинхронные операции или фоновые потоки, чтобы избежать блокировки UI при выполнении длительных действий в приложениях Windows Forms и WPF. Применяйте AJAX для осуществления асинхронных запросов в ASP.NET. Обеспечивайте пользователю обратную связь о ходе выполнения процесса для длительных операций. Предоставьте пользователю возможность отменить длительную операцию.
  - избегайте смешения логики обработки UI и формирования визуального представления;
  - при выполнении ресурсоемких вызовов к удаленным ресурсам или слоям, таких как вызов веб-сервисов или запрос к базе данных, проведите анализ, возможно, будет целесообразным

делать эти запросы к сервису с детализированным интерфейсом (много мелких запросов) либо с обобщенным интерфейсом (один большой запрос). Если пользователь запрашивает большие объемы данных для выполнения операции, первым делом извлекайте лишь то, что необходимо для отображения и начала работы, а затем постепенно догружайте данные в фоновом потоке или по мере необходимости (например, разделение данных на страницы и виртуализация UI).

- Обеспечьте понятные пользователю сообщения об ошибках для уведомления об ошибках приложения, но убедитесь, что не включаете конфиденциальные данные в страницы ошибок, в сообщения об ошибках, файлы журналов. Постарайтесь по возможности привести приложение в согласованное состояние или, если это невозможно, обеспечьте завершение выполнения.
- Разрабатывайте стратегию навигации так, чтобы пользователи могли удобно перемещаться по экранам или страницам приложения, и чтобы можно было отделить функциональность навигации от формирования и обработки UI. Обеспечьте единообразное представление навигационных ссылок и элементов управления во всем приложении, чтобы не запутывать пользователя и скрыть сложность приложения. При проектировании стратегии навигации руководствуйтесь следующими рекомендациями:
  - проектируйте панели инструментов и меню, чтобы пользователи могли быстро находить функции, предоставляемые UI;
  - обеспечьте предсказуемость реализации навигации между формами с помощью мастеров и определитесь с тем, как будете сохранять состояние навигации между сеансами в случае необходимости;
  - избегайте дублирования логики для обработчиков событий навигации и по возможности не указывайте в коде пути переходов, для обработки обычных действий из множества источников используйте шаблон Command (Команда).
- Эффективная стратегия проверки пользовательского ввода и данных имеет критически важное значение для безопасности и корректной работы приложения.
- Определите правила валидации пользовательского ввода и бизнес-правила, существующие в слое представления.
- При проектировании стратегии проверки пользовательского ввода и данных руководствуйтесь следующими рекомендациями:

- Проверка данных при вводе ввода должна проводиться в слое представления, тогда как проверка на соответствие бизнес-правилам – в бизнес-слое.
- Если бизнес-слой и слой представления разнесены физически, логика проверки на соответствие бизнес-правилам должна дублироваться в слое представления для улучшения удобства использования и уменьшения времени отклика. Этого можно достичь с помощью метаданных или путем применения одинаковых компонентов правил проверки в обоих слоях.

Применение компонентов на слое представления (задача 2)

Компоненты являются средством изоляции определенных наборов функций в элементах, которые могут распространяться и устанавливаться отдельно от другой функциональности.

Общие рекомендации по проектированию компонентов включают известные принципы проектирования SOLID, рекомендуется проектировать сильно связные компоненты, компонент не должен зависеть от внутренних деталей других компонентов и от того, как компоненты будут взаимодействовать друг с другом.

Компоненты слоя представления реализуют функциональность, необходимую для обеспечения взаимодействия пользователей с приложением.

Выше было отмечено, что для слоя представления реализуются два типа компонентов: непосредственно пользовательского интерфейса и логики представления. Рассмотрим их более подробно.

- Компоненты пользовательского интерфейса.

В этом случае конкретная реализация пользовательского интерфейса приложения инкапсулирована в компоненты пользовательского интерфейса. Это визуальные элементы приложения, используемые для отображения данных пользователю и приема пользовательского ввода. При реализации шаблона Separated Presentation эти компоненты называют представлениями (Views), и их роль заключается в предоставлении пользователю интерфейса, который обеспечивает наиболее соответствующее представление данных и логики приложения, а также в интерпретации пользовательского ввода и передаче его в компоненты логики представления, которые определяют влияние ввода на данные и состояние приложения. В некоторых случаях в компонентах пользовательского интерфейса может содержаться специальная логика реализации пользовательского интерфейса, однако, как правило, они включают минимальный объем логики приложения, поскольку это может негативно сказаться на удобстве обслуживания и возможности повторного использования, а также усложнить модульное тестирование.

- Компоненты логики представления.

Логика представления – это код приложения, определяющий поведение и структуру приложения таким образом, что они не зависят от какой-либо конкретной реализации пользовательского интерфейса. Компоненты логики представления, главным образом, обеспечивают реализацию вариантов использования приложения (или пользовательских историй) и координируют взаимодействия пользователя с базовой логикой и состоянием приложения независимо от UI. Также они отвечают за организацию поступающих с бизнес-слоя данных в формат, пригодный для потребления компонентами UI. Например, они могут агрегировать данные из многих источников и преобразовывать их для большего удобства отображения.

Компоненты логики представления можно подразделить на две категории:

° Компоненты Presenter, Controller, Presentation Model и ViewModel. Данные типы компонентов используются при реализации шаблона Separated Presentation и часто инкапсулируют логику представления слоя представления. Чтобы обеспечить максимальные возможности повторного использования и удобство тестирования, эти компоненты не привязаны ни к одному конкретному классу, элементу или элементу управления UI.

◦ Компоненты сущностей представления. Эти компоненты инкапсулируют бизнес-логику и данные и упрощают их потребление пользовательским интерфейсом и компонентами логики представления, например, путем преобразования типов данных или агрегации данных из нескольких источников. В некоторых случаях это бизнес-сущности бизнес-слоя, используемые напрямую слоем представления. В других случаях они могут представлять подмножество компонентов бизнес-сущностей и создаваться специально для поддержки слоя представления приложения. Сущности представления помогают обеспечить непротиворечивость и действительность данных в слое представления. В некоторых шаблонах раздельного представления эти компоненты называют моделями.

Компоненты модели представления и сущности представления могут быть полезны при размещении слоя представления и бизнес-слоя на разных уровнях, как это показано на рисунке 5.

При проектировании компонентов пользовательского интерфейса руководствуйтесь следующими рекомендациями:

- Разбейте страницы или окна на отдельные пользовательские элементы управления, чтобы упростить и обеспечить возможность повторного использования этих элементов управления. Выбирайте соответствующие компоненты UI и используйте преимущества возможностей связывания данных элементов управления, применяемых в UI.

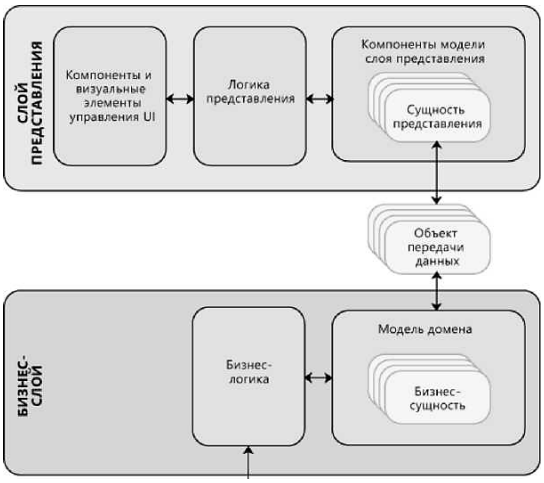


Рис. 5. Компоненты модели представления и сущности представления

- Избегайте создания иерархий наследования пользовательских элементов управления и страниц, чтобы обеспечить возможность повторного использования кода. Отдавайте предпочтение композиции, а не наследованию, и создавайте компоненты логики представления, пригодные для повторного использования.
- Создавайте специализированные элементы управления, только если это необходимо для специализированного отображения или сбора данных. Если видите, что имеющиеся требования к UI не могут быть реализованы стандартными элементами управления, прежде чем создавать собственные специализированные элементы управления, рассмотрите возможность приобретения готового набора элементов управления.
- При создании специализированных элементов управления старайтесь расширять существующие элементы управления, а не создавать элементы управления с нуля. Расширяйте существующие элементы управления путем подключения к ним поведения, а не наследования от них. Реализуйте поддержку дизайнера для специализированных элементов управления, чтобы разработчикам было проще работать с ними.

Компоненты логики представления занимаются невизуальными аспектами пользовательского интерфейса, к которым обычно относится проверка, ответ на действия пользователя, взаимодействие компонентов UI и координирование взаимодействий с пользователем.

Рекомендации по проектированию компонентов логики представления:

- Создавайте эти компоненты, только если собираетесь выполнять в слое представления большой объем обработки, которая должна быть отделена от компонентов UI, или если хотите обеспечить более благоприятные условия для модульного тестирования логики

представления, т.е. если UI требует сложной обработки или должен обмениваться данными с другими слоями, используйте компоненты логики представления для отделения этой обработки от компонентов UI.

- Используйте компоненты логики представления для хранения состояния, относящегося к UI, но не характерного для конкретной реализации. Старайтесь не включать в компоненты логики представления бизнес-логику или бизнес-правила, кроме валидации ввода и данных. Также избегайте реализации в компонентах логики представления логики формирования визуального представления UI или специализированной логики UI.
- С помощью компонентов логики представления обеспечьте согласованное состояние пользовательского интерфейса при восстановлении приложения после сбоя или ошибки.
- В случае необходимости поддержки сложного рабочего процесса создавайте отдельные компоненты рабочего процесса, использующие систему управления процессом, например, такие как Windows Workflow Foundation, или реализуйте собственный механизм в бизнес-слое приложения.

Компоненты модели представления представляют данные, поступающие с бизнес-слоя, в формате, доступном для использования UI и компонентами логики представления. Обычно модели представляют данные и поэтому используют компоненты доступа к данным и, возможно, компоненты бизнес-слоя для сбора этих данных. Если модель также инкапсулирует бизнес-логику, ее обычно называют сущностью представления. Компоненты модели представления могут, к примеру, агрегировать данные из множества источников, преобразовывать данные для обеспечения удобства их отображения в UI, реализовывать логику проверки и помогать в представлении бизнес-логики и состояния в рамках слоя представления. Обычно они используются для реализации шаблонов раздельного представления, таких как MVP или MVC.

При проектировании компонентов модели представления руководствуйтесь следующими рекомендациями:

- Определитесь, нужны ли вам компоненты модели представления. Обычно модели слоя представления используются для отображения специальных данных или форматов слоя представления или в случае применения шаблона раздельного представления, такого как MVP или MVC.
- Работая с элементами управления с привязкой к данным, проектируйте или выбирайте соответствующие компоненты модели представления, которые можно легко связать с элементами управления UI. При использовании в качестве формата компонента модели представления специальных объектов, коллекций или

наборов данных обеспечьте реализацию ими соответствующих интерфейсов и событий для поддержки привязки данных.

- При выполнении валидации данных в слое представления размещайте код валидации в компонентах модели представления. Но также продумайте преимущества использования кода или библиотек для централизованной валидации.
- Рассмотрите требования сериализации для данных, передаваемых в компоненты модели представления, если эти данные будут передаваться по сети или сохраняться на жестком диске клиента.

Также необходимо выбрать подходящий тип данных для компонентов модели представления и сущностей представления. Этот выбор определяется требованиями, предъявляемыми к приложению, и ограничениями, налагаемыми инфраструктурой и возможностями разработки. Начните с выбора формата данных слоя представления и примите решение о том, будут ли компоненты также инкапсулировать бизнес-логику и состояние.

Стратегии обработки ошибок и валидации (задача 3)

Компоненты UI являются внешней границей приложения и поэтому должны реализовывать соответствующую стратегию обработки ошибок для обеспечения стабильности приложения и положительного впечатления при взаимодействии с пользователем.

При проектировании стратегии обработки ошибок рассмотрите следующие варианты [1]:

- Стратегия централизованной обработки исключений. Обработка исключений и ошибок относится к сквозной функциональности. Она должна реализовываться в отдельных компонентах, которые обеспечивают централизацию этой функциональности и делают ее доступной во всех слоях приложения. Это также упрощает обслуживание и способствует повторному использованию.
- Протоколирование исключений. Очень важно протоколировать ошибки на границах системы, чтобы служба поддержки могла выявлять и диагностировать их. Это важно для компонентов представления, но может создавать большие сложности для кода, выполняющегося на клиентских компьютерах. Будьте осторожны и тщательно выбирайте методы протоколирования информации личного порядка (Personally Identifiable Information, PII) или конфиденциальных данных и обратите особое внимание на размер и размещение журнала.
- Вывод на экран понятных пользователю сообщений. При использовании этой стратегии в случае возникновения ошибки на экран выводится понятное пользователю сообщение с указанием причины ошибки и описанием, как ее можно исправить. Например,

ошибки проверки данных должны отображаться так, чтобы было понятно, какие данные являются ошибочными и почему. В этом сообщении также указывается, как пользователь может исправить или ввести действительные данные.

- Разрешение повторной попытки. При использовании этой стратегии на экран выводится понятное пользователю сообщение, объясняющее причину ошибки и предлагающее пользователю повторить операцию. Такая стратегия полезна, если ошибки формируются из-за возникновения временных исключительных ситуаций, таких как недоступность ресурса или истечение времени ожидания сети.
- Вывод на экран универсальных сообщений. Если в приложении возникает непредвиденная ошибка, данные ошибки необходимо запротоколировать, но для пользователя вывести только универсальное сообщение. Предоставьте пользователю уникальный код ошибки, который может быть представлен группе технической поддержке. Эта стратегия полезна при возникновении непредвиденных исключений. Как правило, в случае возникновения непредвиденного исключения рекомендуется закрыть приложение, чтобы предотвратить повреждение данных или риски безопасности.

Эффективная стратегия валидации пользовательского ввода поможет фильтровать нежелательные или злонамеренные данные и будет способствовать повышению защищенности приложения. Как правило, валидация ввода осуществляется слоем представления, тогда как проверка на соответствие бизнес-правилам проводится компонентами бизнес-слоя.

При проектировании стратегии проверки прежде всего необходимо определить все вводимые данные, подлежащие проверке. Например, ввод от веб-клиента в поля формы, параметры (такие как данные операций GET и POST и строки запросов), скрытые поля и состояние представления (view state) подлежат проверке. В общем, проверяться должны все данные, поступающие из источников, не имеющих доверия.

Для приложений, имеющих компоненты и на стороне клиента, и на стороне сервера, таких как приложения RIA или насыщенные клиентские приложения, вызывающие сервисы на сервере приложений, кроме всех проверок на клиенте, должна проводиться дополнительная проверка на сервере. Но для повышения удобства использования и по соображениям производительности некоторые из проверок можно продублировать на клиенте. Проверка на клиенте позволяет обеспечивать пользователям быструю обратную связь в случаях ввода ими некорректных данных. Это может сохранить время и полосу пропускания, но не забывайте, что злоумышленники могут обойти любую реализованную на клиенте проверку.

Определившись с данными, подлежащими проверке, выберите методики проверки для них. Самыми распространенными методиками проверки являются:

- Прием заведомо допустимого (список разрешенного ввода – позитивная проверка). Принимаются только данные, удовлетворяющие заданным критериям, все остальные данные отклоняются.
- Отклонение заведомо недопустимого (Список запрещенного ввода – негативная проверка). Принимаются данные, не содержащие известный набор символов или значений.
- Очистка. Известные плохие символы или значения устраняются или преобразовываются с целью сделать ввод безопасным.

Рекомендуется принимать заведомо допустимые значения (список разрешенного ввода), а не пытаться выявить все возможные недействительные или злонамеренные значения, которые должны быть отклонены. Если невозможно полностью определить список известных допустимых значений, можно дополнить проверку частичным списком известных недопустимых значений и/или проводить очистку в качестве второй линии защиты. Разные технологии представления используют разные подходы к проверке и информированию пользователя о проблемах. Например, в WPF используются конвертеры и объекты правил проверки, подключаемые с помощью XAML, а в Windows Forms проверку обеспечивают события валидации и привязки.

Практические задания

Задача 1. Разработайте предварительную модель слоя представления с учетом пожеланий пользователя, выберите соответствующий исходным данным тип приложения и технологию UI.

Задача 2. Разработайте предложения по применению компонентов на данном слое разрабатываемого приложения. Перечислите, какие задачи эти компоненты должны решать в целях достижения требуемой функциональности.

Задача 3. Разработайте предложения по реализации с помощью компонентов стратегий обработки исключений и валидации.

Отчетность по работе

Отчет должен содержать:

1. Титульный лист.
2. Цель и задачи лабораторной работы.
3. Результаты выполнения практической части.
4. Диаграммы и другие графические материалы.
5. Выводы по работе.

Контрольные вопросы

1. Раскройте понятие слоя представления.
2. Перечислите общие рекомендации по проектированию пользовательского интерфейса.
3. Укажите способы повышения отзывчивости пользовательского интерфейса.
4. Приведите пример понятного пользователю сообщения для его уведомления об ошибках приложения.
5. Приведите пример правила валидации пользовательского ввода и бизнес-правила, существующие в слое представления.
6. Перечислите основные способы обработки ошибок валидации пользовательского ввода.
7. Укажите свое мнение о применимости принципов SOLID при проектировании классов, входящих в компонент пользовательского интерфейса.
8. Перечислите и кратко опишите типы компонентов слоя представления.
9. Опишите основное содержание методики проектирования компонентов представления.
10. Опишите назначение и возможную структуру компонента модели представления, приведите пример.

Лабораторная работа 4. Разработка архитектуры приложения по обработке данных инфокоммуникационной системы

Цель работы

Изучение основных рекомендаций по проектированию слоя доступа к данным приложения, освоение процесса проектирования интерфейса доступа к данным в типовой архитектуре многослойного приложения и приобретение навыков разработки компонентов слоя доступа к данным.

Теоретические основы работы

На рис. 6 показано место слоя доступа к данным в общей архитектуре приложения.

Слой доступа к данным может включать следующие компоненты [1]:

- Компоненты доступа к данным. Эти компоненты абстрагируют логику, необходимую для доступа к базовым хранилищам данных. Они обеспечивают централизацию общей функциональности доступа к данным, что способствует упрощению настройки и обслуживания приложения.

Некоторые инфраструктуры доступа к данным могут требовать, чтобы общая логика доступа к данным была определена и реализована в

отдельных вспомогательных или служебных компонентах доступа к данным, пригодных для повторного использования. Другие инфраструктуры доступа к данным, включая многие инфраструктуры объектно-реляционного сопоставления (Object/Relational Mapping, O/RM), реализуют такие компоненты автоматически, сокращая объем кода доступа к данным, который должен написать разработчик.

- Агенты сервисов. Если компонент бизнес-слоя должен выполнять доступ к данным, предоставляемым внешним сервисом, может понадобиться реализовать код управления семантикой взаимодействия с этим конкретным сервисом.

Агенты сервисов реализуют компоненты доступа к данным, которые изолируют меняющиеся требования вызова сервисов от приложения и могут обеспечивать дополнительные сервисы, такие как кэширование, поддержку работы в автономном режиме и базовое преобразование между форматом данных, предоставляемых сервисом, и форматом, поддерживаемым вашим приложением.

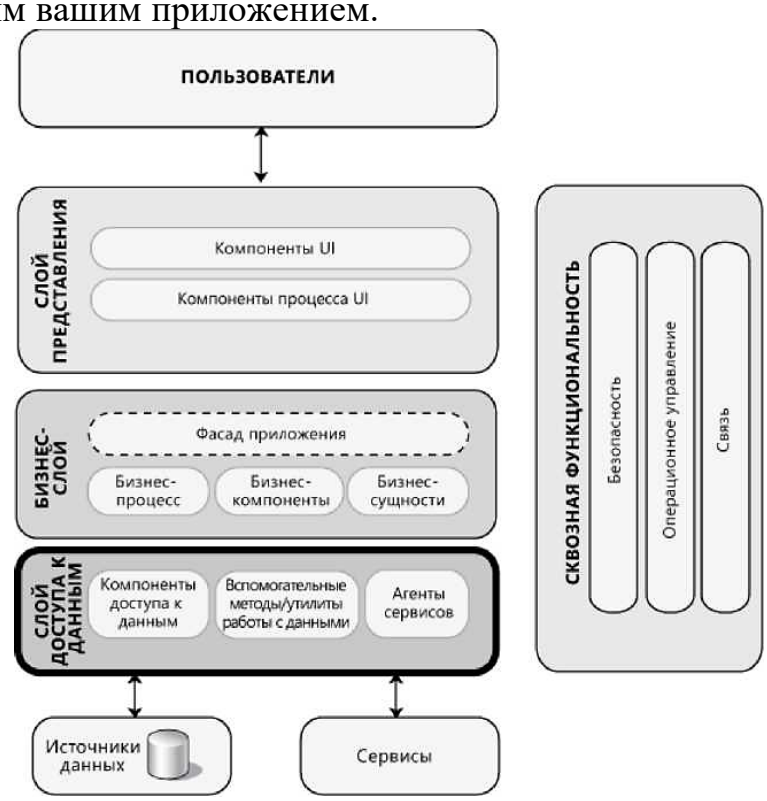


Рис. 6. Слой доступа к данным в типовом приложении

Модель слоя доступа к данным (задача 1)

Требования к слою доступа к данным, которые должны быть обеспечены при его проектировании и реализации, следующие: он должен отвечать требованиям приложения, работать эффективно и безопасно, а также обеспечивать простоту обслуживания и расширения в случае изменения бизнес-требований.

Для обеспечения реализации указанных требований при проектировании слоя доступа к данным следует руководствоваться следующими общими рекомендациями:

- Выбирайте технологию доступа к данным в зависимости от типа данных, с которыми придется работать, и того, как предполагается обрабатывать данные в приложении.

Для каждого сценария существуют наиболее подходящие технологии, все технологии доступа к данным с перечислением преимуществ и предложений относительно применения в том или ином сценарии можно посмотреть в приложении С «Матрица технологий доступа к данным» [1].

- Используйте абстракцию для реализации слабо связанного интерфейса слоя доступа к данным. Этот подход можно реализовать путем определения интерфейсных компонентов, таких как шлюз с общеизвестными входными и выходными параметрами, который преобразует запросы в формат, понятный компонентам слоя.

Кроме того, с помощью интерфейсных типов или абстрактных базовых классов можно определить совместно используемую абстракцию, которая должна быть реализована интерфейсными компонентами.

- Инкапсулируйте функциональность доступа к хранилищу данных в слое доступа к данным – слой доступа к данным должен скрывать детали доступа к источнику данных.

Он должен обеспечивать управление подключениями, формирование запросов и сопоставление сущностей приложения со структурами источника данных. Потребители слоя доступа к данным взаимодействуют с ним через абстрактные интерфейсы с использованием сущностей приложения, таких как, например, объекты предметной области, типизированные наборы данных и XML, и не должны знать внутренних деталей слоя доступа к данным. Такое разделение функциональных областей упрощает разработку и обслуживание приложения.

- Определите вариант сопоставления сущностей приложения со структурами источника данных. Тип сущности, используемой в приложении, является основным фактором при принятии решения о методе сопоставления этих сущностей со структурами источника данных.

Обычно для этого используются шаблоны Domain Model или Table Module либо механизмы объектно-реляционного сопоставления (Object/Relational Mapping, O/RM), однако бизнес-сущности могут быть реализованы с использованием разнообразных подходов. Необходимо определить стратегию заполнения бизнес-сущностей или структур данных из источника данных и их предоставления для доступа с бизнес-слоя или слоя представления приложения.

- Рассмотрите возможность объединения структур данных. При предоставлении данных через сервисы можно использовать объекты

передачи данных (Data Transfer Objects, DTO) для организации данных в унифицированные структуры.

Кроме того, объекты DTO позволяют реализовать слабо детализированные операции, обеспечивая при этом структуру для перемещения данных. Объекты DTO также могут объединять бизнес-сущности при выполнении агрегированных операций.

- Реализуйте безопасное управление подключениями. Слой доступа к данным должен создавать и управлять подключениями ко всем источникам данных, используемым приложением. Необходимо выбрать метод хранения и защиты данных подключения, соответствующий требованиям безопасности предприятия.

Возможными вариантами могут быть шифрование разделов конфигурационного файла или сохранение конфигурационных данных исключительно на сервере.

- Определите, как будут обрабатываться исключения, возникающие при обработке данных. Слой доступа к данным должен перехватывать и обрабатывать (по крайней мере, обеспечивать начальный этап) все исключения, связанные с источниками данных и операциями CRUD (Create, Read, Update и Delete).

Исключения, касающиеся самих данных и доступа к источникам данных, и ошибки истечения времени ожидания должны обрабатываться в этом слое и передаваться в другие слои, только если сбои оказывают влияние на скорость ответа и функциональность приложения.

- Учтите риски безопасности. Слой доступа к данным должен обеспечивать защиту от попыток похищения или повреждения данных и защищать механизмы, используемые для получения доступа к источнику данных.

Рекомендуется очищать детализированную информацию об ошибках или исключениях, чтобы не было возможности разглашения данных из источника данных, и использовать менее привилегированные учетные записи, обладающими только необходимыми для осуществления операций приложения правами. Даже если сам источник данных обладает возможностью ограничивать привилегии, защита должна быть реализована и в слое доступа к данным, и в источнике данных. Доступ к базе данных должен осуществляться через параметризованные запросы, чтобы предотвратить возможность успеха атак с внедрением SQL-кода (SQL-инъекции). Никогда не применяйте конкатенацию строк для построения динамических запросов на основе вводимых пользователем данных.

- Сократите количество сетевых вызовов и обращений к базе данных. Рассмотрите возможность группировки команд в одну операцию базы данных.
- Учтите требования производительности и масштабируемости. Например, для приложения электронной коммерции именно

производительность слоя доступа к данным, скорее всего, будет «узким местом» приложения. Если производительность слоя доступа к данным критична, используйте инструменты профилирования, чтобы понять и затем сократить количество или разбить ресурсоемкие операции с данными.

При проектировании слоя доступа к данным существует ряд общих вопросов, на которые следует обратить внимание, эти вопросы можно сгруппировать по определенным областям проектирования, которые чаще всего вызывают затруднения [1]:

- Пакетная обработка,
- Большие бинарные объекты,
- Подключения,
- Формат данных,
- Управление исключениями,
- Объектно-реляционное сопоставление,
- Запросы,
- Хранимые процедуры,
- Сравнение хранимых процедур и динамического SQL,
- Транзакции,
- Валидация,
- XML.

➢ Пакетная обработка

Пакетная обработка команд базы данных может обеспечить повышение производительности слоя данных, поскольку каждый запрос к среде выполнения базы данных порождает издержки.

Пакетирование может сократить общие издержки за счет увеличения пропускной способности и уменьшения задержки. Пакетирование схожих запросов может повысить производительность, поскольку это позволяет базе данных выполнять кэширование и повторно использовать план выполнения для аналогичного запроса. При проектировании пакетной обработки руководствуйтесь следующими рекомендациями:

• Рассмотрите возможность пакетирования команд для сокращения количества обращений к базе данных и минимизации трафика. Однако наибольший положительный эффект будет иметь пакетирование схожих запросов. Объединение разных или случайных запросов не обеспечивает значительного уменьшения издержек.

• Используйте пакетные команды и DataReader (модуль чтения данных) для загрузки и копирования множества наборов данных. Но при загрузке больших объемов файлов данных в базу данных рекомендуется применять утилиты базы данных для массового копирования.

• Не используйте транзакции для длительных пакетных команд, это может заблокировать ресурсы базы данных.

- **Большие бинарные объекты**

Если данные хранятся и извлекаются как единый поток, их можно рассматривать как большой бинарный объект или BLOB (Binary Large Object). BLOB может иметь внутреннюю структуру, но эта структура скрыта от базы данных, в которой он хранится, или слоя доступа к данным, который выполняет чтение и запись этого объекта. BLOB-данные, как правило, хранятся непосредственно в базе данных или в файловой системе. BLOB-объекты обычно используются для хранения изображений, но также могут использоваться для хранения бинарного представления объектов. При проектировании BLOB-объектов руководствуйтесь следующими рекомендациями:

• Определитесь в необходимости хранения BLOB-данных в базе данных. Современные базы данных обеспечивают намного лучшую обработку BLOB-данных, предоставляя возможность выбора соответствующего типа данных столбца, а также удобство обслуживания и работы, контроль версий и хранение соответствующих метаданных. Тем не менее, подумайте, возможно, практичнее будет хранить BLOB на диске, а в базе данных размещать только ссылку на эти данные.

• Рассмотрите возможность применения BLOB-объектов для упрощения синхронизации больших бинарных объектов между серверами.

• Если необходимо обеспечить возможность поиска в BLOB-данных, предусмотрите в базе данных дополнительные поля для поиска, чтобы не приходилось проводить синтаксический анализ данных в BLOB-полях.

• При извлечении приводите BLOB к соответствующему типу для работы в бизнес- слое или слое представления.

- **Подключения**

Создание и управление подключениями занимает ценные ресурсы как слоя доступа к данным, так и источника данных. Чтобы обеспечить максимальную производительность и безопасность, при проектировании подключений слоя доступа к данным руководствуйтесь следующими рекомендациями:

• Открывайте подключения как можно позже и закрывайте их как можно раньше. Никогда не оставляйте подключения открытыми без надобности.

• По возможности осуществляйте транзакции в одно подключение.

• Применение модели доверенной подсистемы безопасности позволит воспользоваться преимуществами пула подключений – по возможности избегайте олицетворения или использования личных удостоверений.

• В целях безопасности не храните данные подключения в системных или пользовательских данных (Data Source Name (DSN)).

• В случае необходимости предусмотрите логику повторного подключения для обработки ситуаций, когда подключение к источнику данных потеряно либо закрыто по истечении времени ожидания. Однако в случае возникновения некоторых проблем, таких как конкуренция за ресурсы, повторные попытки выполнения операции могут усугублять их, негативно сказываясь на масштабировании.

- **Формат данных**

Формат данных и сериализация важны для хранения и извлечения состояния приложения бизнес-слоем. Правильный выбор формата данных обеспечивает возможность взаимодействия с другими приложениями и способствует обмену сериализованными данными между разными процессами или компьютерами.

При выборе формата данных руководствуйтесь следующими рекомендациями:

• Используйте XML для обеспечения возможности взаимодействия с другими системами и платформами или при работе со структурами данных, которые могут меняться со временем.

• Используйте объекты DataSet для сценариев без постоянного подключения в простых приложениях, выполняющих операции CRUD.

• Если требуется передавать данные через физические границы, обратите внимание на требования сериализации и возможности взаимодействия. Например, продумайте, как будет выполняться сериализация бизнес-объектов, как они будут транслироваться в объекты передачи данных (Data Transfer Objects, DTOs), если есть такая необходимость, и какие форматы может принимать целевой слой.

- **Управление исключениями**

Проектируйте стратегию централизованного управления исключениями, чтобы обеспечить единообразие при перехвате и формировании исключений в слое доступа к данным. По возможности концентрируйте логику обработки исключений в компонентах приложения, реализующих сквозную функциональность. Особое внимание уделяйте исключениям, распространяющимся через границы доверия и на другие слои и уровни. Учитывайте при проектировании и необрабатываемые исключения, чтобы они не приводили к снижению надежности приложения или разглашению конфиденциальных данных приложения. При проектировании стратегии управления исключениями руководствуйтесь следующими рекомендациями:

• Выявите исключения, которые должны перехватываться и обрабатываться слоем доступа к данным. Например, обычно проблемы

взаимоблокировки и подключений могут быть решены в слое доступа к данным. Тем не менее, некоторые исключения, такие как нарушения параллелизма, должны быть представлены пользователю для разрешения возникшей ситуации.

- Выберите соответствующую стратегию распространения исключений. Например, разрешите исключениям распространяться к граничным слоям, где они могут быть зарегистрированы и в случае необходимости преобразованы перед передачей в следующий слой. Применяйте контекстные идентификаторы, это позволит находить взаимосвязанные исключения в разных слоях при проведении анализа основных причин ошибок и сбоев.
- Где это возможно, реализуйте процесс повторного подключения для операций, в которых могут возникать ошибки источника данных или ошибки в результате истечения времени ожидания.
- Убедитесь, что перехватываете исключения, которые не будут перехвачены где- либо в другом месте (например, глобальном обработчике ошибок), и очищайте ресурсы и состояние после возникновения исключения.
- Правильно выберите стратегию протоколирования и уведомления для критических ошибок и исключений, чтобы регистрировать достаточный объем сведений об исключениях и не разглашать конфиденциальных данных.

- **Объектно-реляционное сопоставление**

При проектировании объектно-ориентированного (ОО) приложения учтите несоответствия модели ОО и реляционной модели и факторы, которые могут усложнить задачу по передаче данных между ними. Например, инкапсуляция в ОО-дизайнах, где все поля скрыты, может противоречить открытой природе свойств в базе данных. К другим примерам несоответствия относятся отличия в применяемых типах данных, структуре, транзакциях и механизмах обработки данных.

Существуют два основных средства решения этой проблемы – шаблоны проектирования для доступа к данным, такие как Repository, и инструменты объектно-реляционного сопоставления (Object/Relational Mapping, O/RM).

При проектировании объектно-реляционного сопоставления руководствуйтесь следующими рекомендациями:

- Используйте инфраструктуру, обеспечивающую механизмы объектно-реляционного сопоставления (O/RM) между сущностями предметной области и базой данных. В настоящее время предлагаются O/RM решения, которые значительно упрощают задачу разработчика, избавляя от необходимости реализации большого объема собственного кода.

• При работе над абсолютно новым приложением либо в ситуациях, когда обеспечивается полный контроль над схемой базы данных, применяйте инструменты O/RM для формирования схемы, поддерживающей заданную объектную модель и для обеспечения сопоставления сущностей базы данных и предметной области.

• В ситуациях, когда приходится работать с существующей схемой базы данных, применяйте инструменты O/RM для сопоставления модели предметной области и существующей реляционной модели.

• Если вы работаете с небольшим приложением или не имеете доступа к инструментам O/RM, реализуйте шаблон доступа к данным, такой как Repository. В случае применения шаблона Repository объекты хранилища позволяют работать с сущностями предметной области так, как будто они располагаются в памяти.

• При работе с веб-приложениями или сервисами группируйте сущности и поддерживайте механизмы, которые обеспечат загрузку сущностей предметной области с включением в них только необходимых данных. Такой процесс обычно называют отложенной загрузкой. Это позволяет приложениям обеспечивать более высокую пропускную способность, необходимую для поддержки операций без сохранения состояния, и ограничить использование ресурсов благодаря отсутствию необходимости удерживать в памяти инициализованные модели предметной области для каждого пользователя.

➢ **Запросы**

Запросы формируют основные операции манипуляции с данными в слое доступа к данным и являются механизмом преобразования запросов приложения в CRUD-действия базы данных. Поскольку запросы имеют такое большое значение, они должны быть оптимизированы для обеспечения максимальной производительности и пропускной способности базы данных. При использовании запросов в слое доступа к данным руководствуйтесь следующими рекомендациями:

• Используйте параметризованные SQL-запросы и типизированные параметры, чтобы повысить безопасность и снизить шансы успеха атак с внедрением SQL-кода. Не применяйте конкатенацию строк для построения динамических запросов на базе вводимых пользователем данных.

• Рассмотрите возможность использования объектов для построения запросов. Например, реализуйте шаблон Query Object (Объект запроса) или используйте поддержку параметризованных запросов, предоставляемую ADO.NET. Также оптимизируйте схему базы данных для выполнения запросов.

• При построении динамических SQL-запросов избегайте смешения бизнес-логики с логикой, используемой для формирования SQL-

выражений, поскольку это может сильно усложнить обслуживание и отладку кода.

- **Хранимые процедуры**

В прошлом хранимые процедуры обеспечивали лучшую производительность по сравнению с динамическими SQL-выражениями, однако в настоящее время современные ядра СУБД практически уравняли производительность обработки хранимых процедур и динамических SQL-выражений (использующих параметризованные запросы). И теперь основными факторами при принятии решения об использовании хранимых процедур являются абстракция, удобство обслуживания и среда выполнения.

С точки зрения безопасности и производительности основными рекомендациями для хранимых процедур является использование типизированных параметров и недопущение динамического SQL-кода в хранимых процедурах.

При проектировании хранимых процедур руководствуйтесь следующими рекомендациями:

• Используйте типизированные параметры как входные значения процедуры и выходные параметры для возвращения одиночных значений. Рассмотрите возможность применения XML-параметров или табличных параметров для передачи списков данных или табличных данных. В хранимых процедурах не форматируйте данные для отображения, обеспечьте возвращение соответствующих типов и выполняйте форматирование в слое представления.

• Применяйте параметры или переменные базы данных, если требуется формировать динамический SQL в хранимой процедуре. Однако помните, что использование динамического SQL в хранимых процедурах может негативно сказываться на производительности, безопасности и удобстве обслуживания.

• Избегайте создания временных таблиц при обработке данных. Тем не менее, если временные таблицы должны использоваться, создавайте их в памяти, а не на жестком диске.

• Реализуйте соответствующие стратегии обработки ошибок и возвращайте ошибки, которые код приложения может обработать.

Выбор между хранимыми процедурами и динамическим SQL-кодом заключается, главным образом, в принятии решения об использовании SQL-выражений, динамически генерируемых в коде, либо SQL, реализованного в хранимой в базе данных процедуре.

При выборе между хранимыми процедурами и динамическим SQL необходимо учесть требования абстракции, удобства обслуживания и ограничения среды.

Основное преимущество хранимых процедур в том, что они обеспечивают уровень абстракции для базы данных, а это минимизирует зависимость кода приложения от изменений схемы базы данных. Также упрощается реализация и управление безопасностью, поскольку можно ограничить доступ ко всему, кроме хранимой процедуры, и использовать механизмы безопасности, обеспечивающие детализированную защиту и поддерживаемые большинством баз данных (хотя не забывайте, что это может помешать использовать преимущества пула подключений).

Основное преимущество динамических SQL-выражений в том, что зачастую они считаются более гибкими, чем хранимые процедуры, и могут обеспечить более быструю обработку. Многие инфраструктуры O/RM самостоятельно генерируют динамические запросы, существенно сокращая объем кода, который должен быть написан разработчиками.

Выбирая между хранимыми процедурами и динамическим SQL-кодом, руководствуйтесь следующими рекомендациями:

- Для небольшого приложения с единственным клиентом и несколькими бизнес-правилами динамический SQL-код часто является лучшим выбором.
- Для большого приложения с множеством клиентов продумайте, как обеспечить необходимую абстракцию. Примите решение, где эта абстракция должна находиться: в базе данных в форме хранимых процедур или в слое доступа к данным приложения в форме шаблонов доступа к данным или продуктов O/RM.
- Хранимые процедуры позволяют осуществлять операции с использованием большого количества данных ближе к данным, что может улучшить производительность.
- Использование хранимых процедур для доступа к базе данных позволит максимально сократить зависимость кода приложения от изменений схемы базы данных. Это поможет изолировать и максимально сократить количество изменений в коде приложения при нормализации или оптимизации схемы. Изменения во входных и выходных параметрах хранимой процедуры могут влиять на код приложения, но часто эти изменения можно изолировать в отдельных компонентах, выполняющих доступ к хранимой процедуре. Изолировать и максимально сократить изменения в коде приложения при обновлении схемы помогут и инфраструктуры O/RM.
- Принимая решение об использовании динамических SQL-запросов необходимо понимать, какое влияние будут оказывать изменения в схемах базы данных на приложение. Поэтому следует реализовывать слой доступа к данным таким образом, чтобы он обеспечивал отделение бизнес-компонентов от выполнения запросов базы данных. Такое отделение помогут реализовать такие шаблоны как Query Object и Repository. O/RM-

инструменты позволяют полностью отделить бизнес-компоненты от выполнения запросов к базе данных.

• Подумайте о поддержке процесса отладки. Разработчикам приложения будет проще проводить отладку динамического SQL-кода.

- **Транзакции**

Транзакция – это обмен последовательными данными и связанными с ними действиями, которые рассматриваются как единое целое, с целью выполнить запрос и гарантировать целостность базы данных.

Транзакция считается завершенной, только если обработка всех данных и все действия завершены, тогда изменения соответствующей базы данных становятся постоянными. Транзакции поддерживают отмену (откат) действий в случае возникновения ошибки, что помогает сохранить целостность данных базы данных.

Важно правильно выбрать модель параллельной обработки запросов и определить, как будет осуществляться управление транзакциями.

Существуют модели параллельной обработки запросов с пессимистической и оптимистической блокировкой.

При оптимистической блокировке данные не блокируются, и обновления требуют кода для проверки. Обычно проверяется, не изменились ли данные с момента последнего извлечения, для этого используются временные метки.

При пессимистической блокировке данные блокируются и не могут обновляться другими операциями до снятия блокировки.

При проектировании транзакций руководствуйтесь следующими рекомендациями:

• Используйте транзакции только в случае необходимости. Тщательно продумайте границы транзакции, чтобы обеспечить возможность повторных попыток и композиции. Возможно, для простых запросов явная транзакция не нужна, но вы должны убедиться, что не нарушаете стандартного поведения завершения и изоляции транзакции для базы данных. По умолчанию база данных Microsoft SQL Server выполняет каждое SQL-выражение как отдельную транзакцию (режим автоматического завершения транзакции).

• Транзакции должны быть предельно короткими, чтобы максимально сократить время блокировки. Старайтесь избегать использования блокировки для продолжительных транзакций или при доступе к совместно используемым данным, что может блокировать доступ к этим данным другого кода. Не используйте блокировки взаимоисключающего доступа, что может приводить к конфликтам и взаимным блокировкам.

• Используйте соответствующий уровень изоляции, который определяет, как и когда изменения становятся доступными другим операциям. Необходимо найти компромисс между обеспечением

непротиворечивости данных и частотой конфликтов. Высокий уровень изоляции обеспечит большую непротиворечивость данных, но негативно скажется на параллельной обработке в целом. Более низкий уровень изоляции обеспечит лучшую производительность, снизив количество конфликтов, но и более низкую согласованность данных.

• Если нет возможности завершить или откатить транзакцию или при использовании длительных транзакций, реализуйте компенсационные методы для возвращения хранилища данных в предыдущее состояние в случае сбоя операции в рамках транзакции.

• Если требуется выполнять множество запросов к базе данных, рассмотрите возможность применения множества активных результирующих наборов (multiple active result sets, MARS), что обеспечит поддержку множества результирующих наборов только для пересылки и только для чтения и позволит выполнять множество запросов с использованием одного подключения. MARS могут быть полезны в приложениях с параллельной обработкой множества транзакций.

- **Валидация**

Создание эффективного решения валидации пользовательского ввода и данных имеет решающее значение для безопасности приложения.

Определите правила валидации данных, поступающих с других слоев и от компонентов сторонних производителей, а также из базы данных или хранилища данных. При проектировании стратегии проверки руководствуйтесь следующими рекомендациями:

• Проверяйте все данные, получаемые слоем доступа к данным от всех вызывающих сторон. Гарантируйте правильную обработку значений NULL и фильтрацию недействительных символов.

• При проектировании механизмов валидации учитывайте назначение данных. Например, пользовательский ввод, используемый для создания динамического SQL, должен проверяться на наличие символов или шаблонов, встречающихся в атаках внедрением SQL-кода.

• Возвращайте информативные сообщения об ошибках в случае сбоя валидации.

- **XML**

Расширяемый язык разметки (Extensible Markup Language, XML) обеспечивает возможность взаимодействия и обслуживания структур данных вне базы данных.

По соображениям производительности будьте осторожны с применением XML для очень больших объемов данных. Если требуется обрабатывать большие объемы данных в виде XML, применяйте не схемы на базе элементов, в которых значения данных хранятся как значения элементов, а схемы на базе атрибутов: в них значения данных хранятся в

виде атрибутов, следовательно, такие схемы меньшего размера. При проектировании использования XML руководствуйтесь следующими рекомендациями:

- Для доступа к форматированным XML-данным используйте легковесные методы чтения и записи XML, особенно для очень больших наборов XML-данных. Если требуется взаимодействовать с реляционной базой данных, используйте объекты, которые поддерживают эту функциональность, такие как DataSet ADO.NET. При чтении и записи XML используйте общие настройки для обработки пробелов и комментариев.
- Рассмотрите возможность применения схемы XML для описания форматов и обеспечения проверки данных, хранящихся и передаваемых как XML. Используйте расширенные средства проверки для комплексных параметров данных схемы XML. Но не забывайте, что проверка приведет к снижению производительности.
- Храните XML в типизированных столбцах базы данных, если таковые имеются. Это обеспечит максимальную производительность. Если предполагается, что XML- данные будут регулярно запрашиваться, задайте индексы (если используемая база данных их поддерживает).

Проектирование компонентов слоя доступа к данным (задача 2)

Компоненты слоя доступа к данным обеспечивают доступ к данным, размещенным в рамках системы, и к данным, предоставляемым другими сетевыми системами.

Как правило, слой доступа к данным включает два типа компонентов: компоненты доступа к данным и агенты сервисов.

Компоненты доступа к данным абстрагируют логику, необходимую для доступа к базовым хранилищам данных. Для большинства задач доступа к данным необходима общая логика, которая может быть выделена и реализована в отдельных вспомогательных компонентах, доступных для повторного использования, или подходящей вспомогательной инфраструктуре. Это может упростить компоненты доступа к данным и централизовать логику, что облегчает обслуживание. Остальные задачи, общие для компонентов слоя данных и не относящиеся ни к одному набору компонентов, могут быть реализованы как отдельные служебные компоненты. Вспомогательные и служебные компоненты часто объединяются в библиотеку или инфраструктуру, что облегчает их повторное использование в других приложениях.

Агенты сервисов полезны в случае, когда бизнес-компонент должен использовать функциональность, предоставляемую внешним сервисом, и, вероятно, потребуется реализовать код для управления семантикой взаимодействия с этим конкретным сервисом. Агенты сервисов изолируют специальные аспекты вызова разных сервисов в приложении и могут

обеспечивать дополнительные сервисы, такие как кэширование, поддержка работы в автономном режиме и базовое сопоставление форматов данных, предоставляемых сервисом, и форматов, требуемым приложением.

Основные этапы проектирования компонентов данных:

1. Выбор технологии доступа к данным.

При выборе технологии доступа к данным необходимо учесть тип данных, с которыми предполагается работать, и то, как эти данные будут обрабатываться в приложении. Для каждого конкретного сценария есть наиболее подходящие технологии. Выявление ограничений связанных данных, к которым предполагается выполнять доступ, поможет выбрать соответствующую технологию доступа к данным:

- используйте ADO.NET Entity Framework (EF), если хотите создать модель данных и соотнести ее с реляционной базой данных, соотнести один класс с множеством таблиц, используя наследование или выполнять запросы к реляционным хранилищам, не входящим в семейство продуктов Microsoft SQL Server. EF подойдет, если имеется объектная модель, которую необходимо соотнести с реляционной моделью, используя гибкую схему, и необходима гибкость, обеспечивающая возможность отделения схемы сопоставления от объектной модели;
- используйте ADO.NET Core, если для обеспечения полного управления доступом к данным в приложении необходим низкоуровневый API, если хотите использовать уже сделанные инвестиции в ADO.NET-решения, если используете традиционную логику доступа к данным. Также ADO.NET Core подойдет, если нет необходимости в дополнительной функциональности, предлагаемой другими технологиями доступа к данным или если разрабатываемое приложение должно поддерживать сценарии доступа к данным без постоянного подключения.

2. Выбор стратегии сопоставления и определение подхода к реализации доступа к данным (определение используемых бизнес-сущностей и их формата).

Определившись с требованиями источника данных, необходимо выбрать стратегию заполнения бизнес-объектов или бизнес-сущностей данными из хранилища данных и сохранения данных бизнес-объектов или бизнес-сущностей в хранилище данных:

- используйте инфраструктуру O/RM, обеспечивающую преобразования между сущностями предметной области и базы данных, когда имеете контроль над схемой базы данных;
- в объектно-ориентированном проектировании обычно используется модель предметной области, шаблон, основанный на моделировании сущностей соответственно объектам предметной области.

3. Выбор способа подключения к источнику данных.

Нужно принять решение о том, как будет выполняться подключение компонентов доступа к данным и источнику данных:

- используйте пул подключений и открывайте подключения к источнику данных как можно позже, а закрывайте их как можно раньше, это обеспечит блокировку ресурсов лишь на короткие промежутки времени и сделает их более доступными для других процессов;
- по возможности используйте пакетные команды, что позволит сократить количество обращений к серверу базы данных;
- предпочтительнее использовать аутентификацию Windows, а не аутентификацию SQL Server. При работе с Microsoft SQL Server используйте аутентификацию Windows с доверенной подсистемой. При использовании аутентификации SQL применяйте учетные записи с надежными паролями;
- при использовании SQL-выражений для доступа к источнику данных четко обозначьте границы доверия и применяйте параметризованные запросы, а не конкатенацию строк, это обеспечит защиту от атак через внедрение SQL-кода.

4. Выработка стратегий обработки ошибок источника данных.

На данном этапе должна быть выработана общая стратегия обработки ошибок источников данных:

- все исключения, связанные с источниками данных, должны перехватываться слоем доступа к данным. Исключения, касающиеся самих данных, а также ошибки доступа к источнику данных и истечения времени ожидания, должны обрабатываться в этом слое и передаваться в другие слои, только если эти сбои оказывают влияние на время отклика или функциональность приложения;
- предусмотрите логику повтора попыток для обработки ошибок, возникающих при переходе на другой ресурс в случае сбоя сервера или базы данных. Логика повтора должна перехватывать все ошибки, возникающие при подключении к базе данных или выполнении команд (запросов или транзакций);
- правильно выберите время ожидания для подключения и команды. Задание времени ожидания для подключения или команды, превышающего время ожидания клиента (например, в случае со временем ожидания запроса веб-приложения, браузера или веб-сервера), может привести к тому, что время ожидания запроса клиента истечет до того, как подключение к базе данных будет открыто. Задание недостаточного времени ожидания приведет к тому, что обработчик ошибок начнет выполнять логику повтора попыток. Если время ожидания истекает во время выполнения

транзакции, в случае использования пула подключений ресурсы базы данных могут остаться заблокированными после закрытия подключения. В таких случаях, чтобы закрытое подключение не возвращалось в пул, оно должно удаляться. Это обеспечивает откат транзакции и высвобождение ресурсов базы данных.

Практические задания

Задача 1. Разработайте предварительную модель слоя доступа к данным с учетом разрабатываемого приложения.

Выполните следующие действия:

1. Выберите технологию доступа к данным.
2. Определите, как будет выполняться сопоставление сущностей приложения со структурами источника данных.
3. Выберите способ подключения к источнику данных.
4. Определите, как будут обрабатываться исключения, возникающие при обработке данных.
5. Учтите требования производительности и масштабируемости.
6. Рассмотрите необходимость реализации транзакций для выполнения ответственных операций. Подберите подходящий тип поддержки транзакций.

Задача 2. Разработайте предложения по применению компонентов на данном слое разрабатываемого приложения. Перечислите, какие задачи эти компоненты должны решать в целях достижения требуемой функциональности.

Отчетность по работе

Отчет должен содержать:

1. Титульный лист.
2. Цель и задачи лабораторной работы.
3. Результаты выполнения практической части.
4. Схемы и другие графические материалы.
5. Выводы по работе.

Контрольные вопросы

1. Раскройте понятие слоя доступа к данным.
2. Перечислите, какие элементы включает в себя слой доступа к данным.
3. Укажите назначения агентов сервисов.
4. Сформулируйте понятие транзакции. Как этот объект влияет на решение по проектированию слоя доступа к данным?
5. Какие исключения возможны при работе с данными?
6. Какие могут быть выбраны стратегии распространения исключений?

7. Сформулируйте понятие и назначение объектно-реляционного сопоставления.
8. Перечислите сценарии, при которых наиболее удобно применять модели объектно-реляционного сопоставления.
9. Дайте краткую сравнительную характеристику (с точки зрения применения в сценариях работы с данными) хранимым процедурам и SQL-выражениям.

Курсовая работа. Построение архитектуры программного обеспечения облачного приложения

Каждый обучающийся должен разработать архитектуру для конкретной системы согласно индивидуальному заданию.

- ✓ Замечание: рекомендуется инфокоммуникационную систему оставить ту же, что была в лабораторных работах.

Цель работы

Создание архитектуры программного обеспечения облачного приложения – структурированного решения, отвечающего всем техническим и операционным требованиям и обеспечивающего оптимальные общие атрибуты качества, такие как доступность, производительность, безопасность и управляемость.

Задание

1. В зависимости от типа разрабатываемого решения определите подходящую архитектуру для облачного приложения. Подробно обоснуйте свой выбор.
2. Выберите подходящую технологию для вычислений и хранения данных для вашего приложения.
3. Рассмотрите возможность реализации основных принципов проектирования, обеспечивающих масштабируемость, отказоустойчивость и управляемость приложения.
4. Примените показатели качества ПО для разработки вашего решения требуемого уровня.
5. Проанализируйте шаблоны проектирования ПО и примените те, которые лучше всего подходят для вашей задачи.

Методические указания

В процессе решения задач следует учесть особенности применения облачных приложений, поскольку они меняют принципы проектирования приложений (реализованные ранее в лабораторных работах):

- Облачные приложения не монолитны, а состоят из небольших децентрализованных служб.
- Службы взаимодействуют между собой через API с помощью асинхронного обмена сообщениями или событиями.
- Приложения можно масштабировать горизонтально, добавляя новые экземпляры, если появляется потребность в этом.

Далее приводятся методические указания по каждому пункту задания.

Выбор архитектуры для приложения в зависимости от типа разрабатываемого решения

Выбор архитектуры зависит от сложности приложения, области применения, его типа (IaaS или PaaS) и задач, для решения которых оно предназначено. В дальнейшем принятое решение накладывает определенные ограничения на структуру приложения, ограничивая выбор технологий и других элементов приложения. С этими ограничениями связаны как преимущества, так и недостатки выбранной архитектуры. Приведенная далее информация поможет вам найти баланс между ними при реализации той или иной архитектуры.

Исследуйте возможность применения предлагаемых вариантов архитектуры, которые обычно используются в облачных приложениях [4, 5], обратите внимание на описание и логическую схему архитектуры, рекомендации по области применения этой архитектуры, преимущества, недостатки и рекомендации по использованию, рекомендуемый вариант развертывания с использованием подходящих облачных служб.

N-уровневая архитектура чаще всего используется в корпоративных приложениях. Для управления зависимостями приложение делится на слои, каждый из которых отвечает за определенную логическую функцию, например за представление данных, бизнес-логику или доступ к данным. Слой может обращаться с вызовами к другим слоям, расположенным ниже. Однако такое деление на горизонтальные слои может стать причиной возникновения дополнительных трудностей. Например, может оказаться сложно внести изменения в одну часть приложения, не затронув другие его элементы. Поэтому часто обновлять такое приложение непросто, и разработчикам придется реже добавлять новые функции. N-уровневая архитектура может был вариантом при переносе уже используемых приложений, построенных на базе многоуровневой архитектуры и, поэтому такая архитектура чаще всего используется в решениях IaaS.

При **архитектуре web-queue-worker** (веб-интерфейс – очередь – рабочая роль) приложение имеет веб-интерфейс, который обрабатывает HTTP-запросы, и серверную рабочую роль, отвечающую за операции, которые выполняются длительное время или требовательны к

вычислительным ресурсам. Для взаимодействия между интерфейсом и серверной рабочей ролью используется очередь асинхронных сообщений. Данная архитектура подходит для относительно простых задач, требовательных к вычислительным ресурсам. Как и N-уровневая архитектура, эта модель проста для понимания. Использование управляемых служб упрощает развертывание и эксплуатацию. Но при создании приложений для сложных предметных областей бывает трудно контролировать зависимости. Веб-интерфейс и рабочая роль могут легко разрастись до крупных монолитных компонентов, которые трудно поддерживать и обновлять. Как и для N-уровневой архитектуры, для этой модели характерны меньшая частота обновлений и ограниченные возможности совершенствования. Подходит для решений PaaS.

Архитектура микрослужб – микросервисная архитектура (MSA) – приложение состоит из множества небольших независимых служб. Каждая служба отвечает за отдельную бизнес-функцию, причем службы слабо связаны между собой и для взаимодействия используют контракты API. Службы могут развертываться без сложной координации между разработчиками, что упрощает их регулярное обновление. Архитектура микрослужб сложнее в реализации и управлении по сравнению с предыдущими двумя подходами, она требует зрелой культуры управления процессом разработки. При правильной организации процесса разработки такой подход помогает увеличить периодичность выпуска новых версий, ускорить внедрение инноваций и сделать архитектуру более отказоустойчивой.

Архитектура CQRS (Command and Query Responsibility Segregation, распределение ответственности между командами и запросами) позволяет разделить операции чтения и записи между отдельными моделями. В результате части системы, отвечающие за изменение данных, изолируются от частей системы, которые отвечают за чтение данных. Более того, операции чтения могут выполняться в материализованном представлении, которое физически отделено от базы данных, в которую ведется запись. Это позволяет независимым образом масштабировать процессы чтения и записи и оптимизировать материализованное представление для выполнения запросов. Модель CQRS лучше всего использовать для подсистемы более крупной архитектуры. В общем случае ее не стоит применять ко всему приложению, поскольку это неоправданно усложнит его архитектуру. Она хорошо проявляет себя в системах совместной работы, где большое число пользователей одновременно работают с одними и теми же данными.

Архитектура на основе событий использует модель публикации-подписки, в которой поставщики публикуют события, а потребители подписываются на них. Поставщики независимы от потребителей, а потребители независимы друг от друга. Эта архитектура хорошо подходит

для приложений, которые должны быстро получать и обрабатывать большие объемы данных с низкой задержкой, например для Интернета вещей. Кроме того, такая архитектура хорошо проявляет себя в случаях, когда различные подсистемы должны по-разному обрабатывать одни и те же данные о событиях.

Следует учитывать, что при проектировании решения архитектура выступает в роли ограничения, в частности, она определяет, какие элементы можно использовать и какие связи между ними возможны. Ограничения задают конкретный вид архитектуры и позволяют сделать выбор из более узкого набора вариантов. Если ограничения выбранной архитектуры соблюдены, у решения появляются характерные для этой архитектуры свойства.

Например, из анализа архитектуры микрослужб можно выделить следующие ограничения: каждая служба отвечает за отдельную функцию, службы независимы друг от друга, и данные доступны только для той службы, которая отвечает за них, также службы не обмениваются данными. В итоге следование этим ограничениям приводит к созданию системы, в которой службы можно развертывать независимо друг от друга, неисправности изолированы, возможны частые обновления, а в приложение легко добавляются новые технологии.

Прежде чем выбрать архитектуру, убедитесь, что вы хорошо понимаете лежащие в ее основе принципы и связанные с этим ограничения. В противном случае вы можете получить решение, которое внешне соответствует выбранной модели архитектуры, но в нем полностью не раскрыт потенциал этой модели. Важно также руководствоваться здравым смыслом. Иногда разумнее отказаться от того или иного ограничения, чем стремиться к чистоте архитектуры.

Выбор подходящих технологий для вычислений и хранения данных

Следующим решением на начальном этапе проектирования облачного приложения должен быть выбор технологий для вычислений и для хранения данных, поскольку от них зависит общий вид архитектуры приложения.

Выбор технологий для вычислений связан с определением модели размещения вычислительных ресурсов, на которых запускается приложение. В общем виде [4] его можно сформулировать как выбор между моделями IaaS (инфраструктура как услуга), PaaS (платформа как услуга), FaaS (функция как услуга) и всеми промежуточными вариантами. Чтобы принять решение, оцените функции и ограничения конкретной службы, ее доступность и масштабируемость, стоимость и возможности для управления процессом разработки.

При выборе технологий для хранения данных необходимо понять, с какими данными работает ваше приложение, какие данные оно получает и генерирует и какие данные создают его пользователи. Наиболее распространенные типы данных: бизнес-данные, кэши, данные Интернета вещей и телеметрические данные, а также неструктурированные данные. Как правило, приложения работают с несколькими типами данных, и поскольку у каждого типа данных есть свои требования к обработке, вам нужно выбрать подходящее хранилище для каждого из них. Некоторые технологии хранения данных поддерживают различные модели хранения. На основании приведенной в [2] информации выберите модель хранения данных, которая в наибольшей степени соответствует вашим требованиям. После этого определите конкретное хранилище данных в соответствующей категории, исходя из таких параметров, как функциональные возможности, стоимость и удобство управления.

Реализация высокоуровневых принципов проектирования, обеспечивающих масштабируемость, отказоустойчивость и управляемость приложения

В [4] приведены десять принципов проектирования, которые следует иметь в виду при разработке масштабируемого, отказоустойчивого и управляемого приложения:

- Автоматическое устранение неполадок – самовосстановление. Поскольку в распределенных системах часто происходят различного рода сбои, приложение необходимо проектировать так, чтобы оно могло автоматически восстанавливать свою работу при возникновении этих сбоев.
- Принцип избыточности – обеспечивайте резервирование всех компонентов. Резервирование позволяет избежать появления в приложении единственной точки отказа, но учитывайте, что уровень резервирования, заложенный в систему, влияет как на ее стоимость, так и на ее сложность.
- Минимизация координации – сводите к минимуму потребности во взаимодействии компонентов приложения друг с другом. Сведение к минимуму координации между службами приложения помогает обеспечить масштабируемость.
- Горизонтальное масштабирование – учитывайте возможность расширения. Проектируйте приложение так, чтобы оно могло горизонтально масштабироваться, т.е. при необходимости можно было добавлять новые экземпляры. Основным преимуществом облачных решений является гибкое масштабирование – способность использовать столько ресурсов, сколько требуется в данный момент, добавляя новые экземпляры при необходимости и отключая их, когда нагрузка уменьшается.

- Секционирование как средство для обхода ограничений. Секционирование позволяет обходить ограничения, связанные с базами данных, сетевыми и вычислительными ресурсами. Ограничения могут устанавливаться по количеству ядер, размеру базы данных, числу запросов в единицу времени и пропускной способности сети.
- Разработка с учетом эксплуатации – приложение должно проектироваться так, чтобы его было удобно эксплуатировать и у группы эксплуатации были необходимые инструменты. Такие средства, как трассировка процессов, мониторинг загруженности ресурсов, ведение журналов и метрик приложения позволяют в режиме реального времени и в режим анализа постфактум наблюдать за поведением приложения, выявляя условия и причины нестандартного поведения.
- Использование управляемых служб. По возможности используйте подход PaaS (платформа как услуга), а не IaaS (инфраструктура как услуга). Настраивать и администрировать управляемые службы проще: не нужно подготавливать виртуальные машины, настраивать виртуальные сети, управлять исправлениями и обновлениям и делать работу, связанную с запуском программ на виртуальных машинах.
- Соответствие технологии хранения данных – выбирайте технологии хранения данных в соответствии с типом данных и особенностями их использования в приложении.
- Разработка с прицелом на будущее – проектируйте с учетом дальнейшего развития. Все успешные приложения со временем должны развиваться. Чтобы приложение могло развиваться, необходимо закладывать возможности развития на этапе проектирования – быть открытым к расширению.
- Разработка по потребностям – ничего лишнего. Проектируйте приложения с учетом потребностей бизнеса: каждое принятое на этапе проектирования решение должно быть обусловлено конкретным бизнес-требованием.

Применение показателей качества ПО, помогающих добиться решения задачи

Следует учесть, что создание надежного приложения в облаке отличается от создания надежного приложения в корпоративной среде. Возможно, в прошлом при увеличении нагрузки вам приходилось закупать более производительное оборудование. В облачной среде вам нужно будет просто увеличивать число экземпляров. Благодаря использованию стандартного оборудования расходы на облачные среды остаются достаточно низкими. Но вместо того, чтобы предотвращать сбои и снижать «среднее время между сбоями», в этой среде гораздо важнее уменьшать

среднее время восстановления. Ваша задача — свести к минимуму последствия отказа.

Рассмотрите в [4] сведения о разработке отказоустойчивых приложений, изучите понятие «отказоустойчивость» в применении к облачным приложениям и рассмотрите возможные пути достижения отказоустойчивости благодаря применению системного подхода на протяжении всего жизненного цикла приложения от проектирования и разработки до развертывания и эксплуатации.

Рассмотрите в [4, 5] также и другие не менее важные показатели качества:

- Масштабируемость – системы следует создавать с возможностью масштабирования (способность системы обрабатывать увеличенную нагрузку) путем добавления новых экземпляров, например виртуальных машин или реплик базы данных (горизонтальное масштабирование). Горизонтальное масштабирование дает значительные преимущества по сравнению с вертикальным (расширение емкости ресурса, например, использование виртуальной машины большего размера).
- Доступность – определяет целевой уровень обслуживания (SLO), который четко задает ожидаемый уровень доступности (временной период, в течение которого система работает) и способ его измерения. Для определения используйте критический путь. Дополнительный процент доступности может вылиться в несколько часов или дней безотказной работы в год.
- Управление – автоматизируйте развертывание и реализуйте быстрый стандартный процесс для ускорения выпуска новых функций или исправлений ошибок. Предусмотрите операции перехода к предыдущей или следующей версии, а также возможности мониторинга и диагностики.
- Безопасность – не полагайтесь в вопросах безопасности полностью на облачную платформу, которая обеспечивает защиту от различных угроз, таких как сетевые вторжения и самостоятельно реализуйте возможности управления удостоверениями, защиты инфраструктуры, обеспечения суверенитета и шифрования данных в вашем приложении и процедурах управления процессом разработки.

Применение шаблонов проектирования, которые лучше всего подходят для конкретной задачи

Очевидно, шаблоны проектирования помогают создавать надежные, масштабируемые и безопасные приложения в облаке.

Изучите рассматриваемые в [4] шаблоны проектирования. Для каждого шаблона представлены решаемая им задача, рекомендации по применению шаблона и пример, основанный на Microsoft Azure. Для

большинства шаблонов доступны примеры или фрагменты кода, в которых показана реализация шаблона на платформе Azure. Однако большинство шаблонов подходят для любых распределенных систем, размещенных в Azure или на других облачных платформах.

Обратите внимание на наиболее важные из них.

Реализация доступности. На доступность оказывают влияние ошибки системы, проблемы инфраструктуры, вредоносные атаки и нагрузка на систему. Обычно она измеряется в процентах от времени бесперебойной работы. Облачные приложения обычно предоставляют пользователям соглашение об уровне обслуживания. Это означает, что приложения должны быть спроектированы и реализованы для обеспечения максимальной доступности.

Шаблоны:

- Мониторинг работоспособности конечных точек – реализуйте функциональные проверки в приложении, доступном внешним инструментам через открытые конечные точки с регулярными интервалами.
- Балансировка нагрузки на основе очередей – используйте очередь, действующую как буфер между задачей и службой, которая вызывает ее для периодически возникающих высоких нагрузок.
- Регулирование – контролируйте потребление ресурсов, используемых экземпляром приложения, отдельным клиентом или всей службой.

Управление данными — ключевой элемент облачных приложений, который влияет на большинство показателей качества. Данные обычно размещается в разных местах и на нескольких серверах для обеспечения производительности, масштабируемости и доступности, и это может вызывать целый ряд проблем. Например, необходимо поддерживать согласованность данных, а также синхронизировать данные в разных ресурсах.

Шаблоны:

- Отдельный кэш – загрузка данных по требованию в кэш из хранилища данных.
- CQRS – разделение операций, которые считывают данные, от операций, которые обновляют данные, за счет использования отдельных интерфейсов.
- Отслеживание источников событий – использование хранилища, поддерживающего только добавление данных, для записи полного ряда событий, которые описывают действия с данными в домене.
- Таблица индексов – создание индексов для полей в хранилищах данных, на которые часто ссылаются запросы.

- Материализованное представление – создание предварительно заполненных представлений для данных в одном или нескольких хранилищах, если данные имеют неидеальный формат для требуемых операций запроса.
- Сегментирование – разделение хранилища данных на горизонтальные разделы или сегменты.

Проектирование и реализация. Для качественной разработки требуются последовательность и согласованность при проектировании и развертывании компонентов, возможности обслуживания для упрощения администрирования и разработки, а также возможность повторного использования компонентов и подсистем в других приложениях и сценариях использования. Решения, принятые на этапе проектирования и реализации, оказывают огромное влияние на качество и общую стоимость владения облачных приложений и служб.

Шаблоны:

- Посредник – создание вспомогательных служб, которые отправляют сетевые запросы от имени службы или приложения.
- Серверные системы для интерфейсов – создание отдельных серверных служб, которые будут использоваться клиентскими приложениями или интерфейсами.
- CQRS – разделение операций, которые считывают данные, от операций, которые обновляют данные, за счет использования отдельных интерфейсов.
- Консолидация вычислительных ресурсов – консолидация нескольких задач или операций в одном вычислительном блоке.
- Внешнее хранилище конфигураций – перемещение сведений о конфигурации из пакета развертывания приложения в централизованное расположение.
- Агрегирование шлюза – использование шлюза для объединения нескольких отдельных запросов в один запрос.
- Расширение – развертывание компонентов приложения в отдельном процессе или контейнере для изоляции и инкапсуляции.

Обмен сообщениями. Из-за распределенного характера облачным приложениям требуется инфраструктура обмена сообщениями, который связывает компоненты и службы, желательно со слабым уровнем связи, для повышения уровня масштабируемости. Асинхронный обмен сообщениями широко используется и предоставляет множество преимуществ, но также приносит проблемы, такие как упорядочение сообщений, управления подозрительными сообщениями, идемпотентность и т. д.

Шаблоны:

- Конкурирующие потребители – предоставление возможности одновременной обработки сообщений, полученных в том же канале обмена сообщениями, несколькими потребителями.
- Каналы и фильтры – декомпозиция задачи, которая выполняет сложную обработку, на ряд отдельных элементов, которые могут быть использованы повторно.

Управление и мониторинг. Поскольку облачные приложения работают в удаленном центре обработки данных, вы не можете контролировать инфраструктуру или, в некоторых случаях, операционную систему. Из-за этого управление и мониторинг могут оказаться сложнее, чем в локальных средах. Приложения должны предоставлять сведения о времени выполнения, которые администраторы и операторы могут использовать для управления системой и ее мониторинга, а также для поддержки меняющихся бизнес-требований и настройки без остановки и повторного развертывания приложения.

- Посредник – создание вспомогательных служб, которые отправляют сетевые запросы от имени службы или приложения.
- Внешнее хранилище конфигураций – перемещение сведений о конфигурации из пакета развертывания приложения в централизованное расположение.
- Агрегирование шлюза – использование шлюза для объединения нескольких отдельных запросов в один запрос.
- Мониторинг работоспособности конечных точек – реализация функциональных проверок в приложении, доступные внешним инструментам через открытые конечные точки с регулярными интервалами.
- Расширение – развертывание компонентов приложения в отдельном процессе или контейнере для изоляции и инкапсуляции.

Производительность – показатель скорости реагирования системы для выполнения любых действий в течение заданного интервала времени, в то время как **масштабируемость** – способность системы обрабатывать увеличенную нагрузку без снижения производительности или готовность увеличивать объем доступных ресурсов. Облачные приложения обычно сталкиваются с переменными рабочими нагрузками и пиками активности. Их прогнозирование, особенно в многопользовательских сценариях, почти невозможно. Вместо этого приложения должны поддерживать горизонтальное масштабирование в определенных пределах для обработки пиковой нагрузки и масштабирования при ее снижении. Масштабируемость касается не только вычислительных экземпляров, но и других элементов, таких как хранилище данных, инфраструктура обмена сообщениями и т. д.

Шаблоны:

- Отдельный кэш – загрузка данных по требованию в кэш из хранилища данных.
- CQRS – разделение операций, которые считывают данные, от операций, которые обновляют данные, за счет использования отдельных интерфейсов.
- Отслеживание источников событий – использование хранилища, поддерживающего только добавление данных, для записи полного ряда событий, которые описывают действия с данными в домене.
- Таблица индексов – создание индексов для полей в хранилищах данных, на которые часто ссылаются запросы.
- Материализованное представление – создание предварительно заполненных представлений для данных в одном или нескольких хранилищах, если данные имеют не идеальный формат для требуемых операций запроса.
- Очередь на основе приоритетов – назначение приоритетов запросам, которые отправлены службам, для первоочередного получения и обработки запросов с более высоким приоритетом.
- Балансировка нагрузки на основе очередей – использование очереди, действующей как буфер между задачей и службой, которая вызывает ее, для периодически возникающих высоких нагрузок.
- Сегментирование – разделение хранилища данных на горизонтальные разделы или сегменты.
- Размещение статического содержимого – развертывание статического содержимого в службе облачного хранилища, которая может доставить данные напрямую клиенту.
- Регулирование – контроль потребления ресурсов, используемых экземпляром приложения, отдельным клиентом или всей службой.

Выводы

В выводах по итогам выполнения курсовой работы необходимо указать результаты работы и практические рекомендации по проектированию архитектуры облачного приложения.

В качестве рекомендаций можно представить общий шаблон, описывающий структуру выводов (вы можете оформить свои выводы на свое усмотрение):

В работе представлены модель архитектуры облачного приложения…. и перечень технологий, выбранные для разработки облачного приложения …., а также приведено обоснование этого выбора.

Основой архитектуры приложения выбрана … архитектура, как наиболее отвечающая требованиям к приложению, имеющему

независимые компоненты по работе с разными группами пользователей и со внешними сервисами.

Для реализации непосредственно … архитектуры была выбрана служба …, представляющая среду выполнения для …, а также позволяющая отслеживать состояния ….

Хранилище данных в приложении реализуется за счет … баз данных, распределенных по … и тем самым обеспечивающих … и повышенную отказоустойчивость приложения в целом.

В ходе работы были рассмотрены шаблоны проектирования, сгруппированные по направленности применения и приведены некоторые из них как наиболее подходящие для приложения …. Однако, их количество достаточно велико, и многие шаблоны могут быть также применены целиком или частично в зависимости от их преимуществ и недостатков.

Список литературы

1. Microsoft Application Architecture Guide, 2nd Edition. Режим доступа: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650706(v=pandp.10) .

2. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. — 352 с

3. Купер А., Рейман Р., Кронин Д. Алан Купер об интерфейсе. Основы проектирования взаимодействия. – Пер. с англ. – СПб.: СимволПлюс, 2009. – 688 с.

4. Руководство по архитектуре облачных приложений. Microsoft Press. 2017 г. – 329 с.

5. Руководство по архитектуре приложений Azure. Режим доступа: https://docs.microsoft.com/ru-ru/azure/architecture/guide/ .

Осипов Никита Алексеевич

Архитектура программного обеспечения инфокоммуникационных систем

УЧЕБНОЕ ПОСОБИЕ

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49, литер А.