HexProperty Backend Architecture Overview
Version: 1.5
Date: 2024-11-15 11:44

1. Version History
- v1.5 Changes:
  - Added more detailed explanations and implementation details for each architectural layer
  - Expanded the cross-cutting configuration management and stack switching mechanisms
  - Included specific code examples and performance/security considerations
  - Reorganized the table of contents for better flow and comprehensive coverage
- Previous Versions:
  - v1.4: Initial backend architecture documentation

2. Architecture Overview
2.1 Core Principles
- Microservices-based architecture
- Event-driven communication
- Separation of concerns
- Scalability and high availability
- Vendor-independence and configuration-based switching

2.2 System Layers
The HexProperty backend is designed using an 8-layer architecture:

1. Domain Layer
2. Microservices Layer
3. Application Layer
4. Event Infrastructure Layer
5. Data Infrastructure Layer
6. Support Infrastructure Layer
7. Interface Layer
8. Service Mesh Layer

2.3 Cross-Cutting Concerns
- Seamless configuration-based switching between primary and secondary technology stacks
- Centralized configuration management and environment-specific settings
- Observability and monitoring across all layers
- Security and identity management throughout the system

2.4 Configuration Management
The HexProperty backend employs a centralized configuration management system that allows for seamless switching between the primary and secondary technology stacks. This is achieved through the use of environment-specific configuration files, feature flags, and a dedicated configuration service.

3. Layer-by-Layer Architecture
3.1 Domain Layer
- Responsible for encapsulating the core business logic and domain models
- Implemented using a DDD (Domain-Driven Design) approach
- Decoupled from infrastructure concerns and technology-specific details

Implementation Details:
- The domain layer is implemented using Python dataclasses and value objects
- Domain models are designed to be self-validating and immutable
- Domain services handle the orchestration of business logic

Switching Mechanism:
- The domain layer is implemented in a technology-agnostic manner, with no direct dependencies on the underlying infrastructure
- Configuration-based switching is achieved by injecting the appropriate service implementations (e.g., repositories, event publishers) at runtime

3.2 Microservices Layer
- Consists of individual microservices, each responsible for a specific subdomain
- Microservices are self-contained and loosely coupled

- Microservices communicate with each other using the service mesh layer

Implementation Details:
- Microservices are implemented using Google Cloud Run (primary) or Docker containers (secondary)
- Each microservice has its own data persistence, scaling, and deployment strategy
- Microservices leverage the service mesh for service discovery, load balancing, and secure communication

Switching Mechanism:
- The microservices are configured to use the service mesh layer for all inter-service communication
- Switching between the primary (GCP-based) and secondary (Docker-based) stack is achieved by updating the service mesh configuration
- Microservice-specific configuration, such as resource limits and scaling policies, are managed through environment-specific configuration files

3.3 Application Layer
- Provides the entry point for the backend system
- Handles API requests, authentication, and authorization
- Orchestrates the execution of business logic across multiple microservices

Implementation Details:
- The application layer is implemented using Google Cloud Functions (primary) or FastAPI (secondary)
- Authentication and authorization are handled using Google Cloud Identity-Aware Proxy (IAP) or a custom implementation
- The application layer acts as the gateway, routing requests to the appropriate microservices

Switching Mechanism:
- The application layer is configured to use the appropriate service mesh endpoints and authentication providers based on the selected technology stack
- Configuration-based switching is achieved by updating the API gateway and authentication provider settings

3.4 Event Infrastructure Layer
- Responsible for handling asynchronous events and message-based communication
- Provides reliable and scalable event processing capabilities

Implementation Details:
- The event infrastructure layer is built on top of Google Cloud Pub/Sub (primary) or Apache Kafka (secondary)
- Event publishing and consumption are managed through the service mesh, ensuring reliable and secure message delivery
- Event processors are implemented as serverless functions or managed services, depending on the selected technology stack

Switching Mechanism:
- The event infrastructure layer is configured to use the appropriate message bus and event processing services based on the selected technology stack
- Configuration-based switching is achieved by updating the event bus connection details and event processor deployments

3.5 Data Infrastructure Layer
- Provides the data storage and caching capabilities for the system
- Supports both write-optimized and read-optimized data stores

Implementation Details:
- The write-optimized data store is implemented using Google Cloud Datastore (primary) or PostgreSQL (secondary)
- The read-optimized data store is implemented using Google Cloud Spanner (primary) or ClickHouse (secondary)
- Distributed caching is provided by Google Cloud Memorystore (primary) or Redis (secondary)

Switching Mechanism:
- The data infrastructure layer is configured to use the appropriate database and caching services based on the selected technology stack
- Configuration-based switching is achieved by updating the connection details and query configurations for the data stores and caching services

3.6 Support Infrastructure Layer
- Provides auxiliary services that support the core functionality of the system
- Includes logging, metrics, tracing, and other cross-cutting concerns

Implementation Details:
- Logging is provided by Google Stackdriver Logging (primary) or Elasticsearch (secondary)
- Metrics are collected and analyzed using Google Stackdriver Monitoring (primary) or Prometheus (secondary)
- Distributed tracing is implemented using Google Stackdriver Trace (primary) or Jaeger (secondary)

Switching Mechanism:
- The support infrastructure layer is configured to use the appropriate logging, metrics, and tracing services based on the selected technology stack
- Configuration-based switching is achieved by updating the service endpoints and data collection settings for the support services

3.7 Interface Layer
- Handles the external-facing APIs and communication channels
- Provides the entry point for client applications and external integrations

Implementation Details:
- The API Gateway is implemented using Google Cloud API Gateway (primary) or Kong (secondary)
- Authentication and authorization are handled using Google Cloud IAP (primary) or a custom solution (secondary)
- Middleware and request processing are implemented using Google Cloud Functions (primary) or FastAPI (secondary)

Switching Mechanism:
- The interface layer is configured to use the appropriate API gateway, authentication provider, and middleware services based on the selected technology stack
- Configuration-based switching is achieved by updating the gateway configuration, authentication settings, and middleware deployments

3.8 Service Mesh Layer
- Provides the communication backbone for the entire system
- Handles service discovery, load balancing, circuit breaking, and secure communication

Implementation Details:
- The service mesh is implemented using Google Cloud Service Mesh (primary), which is based on Istio
- The service mesh is responsible for all inter-service communication, including synchronous and asynchronous patterns

Switching Mechanism:
- The service mesh configuration is centrally managed and can be switched between the primary (GCP-based) and secondary (Istio-based) implementations
- This switch is achieved by updating the service mesh configuration, which includes updating the proxy sidecar deployments and the control plane settings

4. Technology Stack
4.1 Primary Stack (GCP-based)
- Compute: Google Cloud Run
- Application Frameworks: Google Cloud Functions, Cloud Run
- Event Bus: Google Cloud Pub/Sub
- Event Store: Google Cloud Datastore, Google Cloud Spanner
- Write Database: Google Cloud Datastore, Google Cloud Spanner
- Read Database: Google Cloud Spanner
- Distributed Cache: Google Cloud Memorystore (Redis)

- Logging: Google Stackdriver Logging
- Metrics: Google Stackdriver Monitoring
- Tracing: Google Stackdriver Trace
- API Gateway: Google Cloud API Gateway
- Authentication: Google Cloud Identity-Aware Proxy (IAP)
- Service Mesh: Google Cloud Service Mesh (based on Istio)

4.2 Secondary Stack (Vendor-Independent)
- Compute: Docker containers
- Application Frameworks: FastAPI
- Event Bus: Apache Kafka
- Event Store: PostgreSQL, Elasticsearch
- Write Database: PostgreSQL
- Read Database: ClickHouse
- Distributed Cache: Redis
- Logging: Elasticsearch, Kibana
- Metrics: Prometheus, Grafana
- Tracing: Jaeger
- API Gateway: Kong
- Authentication: Custom solution
- Service Mesh: Istio

4.3 Configuration-Based Switching
The HexProperty backend is designed to allow seamless switching between the primary (GCP-based) and secondary (vendor-independent) technology stacks. This is achieved through a centralized configuration management system that handles the following:

1. **Environment-Specific Configuration**: Environment-specific settings, such as connection details, resource limits, and feature flags, are stored in a centralized configuration repository.
2. **Service Registration and Discovery**: The service mesh layer is responsible for managing service registration and discovery, allowing the seamless switching of service endpoints.
3. **Dependency Injection**: The application uses a dependency injection framework to allow the dynamic injection of service implementations based on the selected technology stack.
4. **Infrastructure as Code**: The entire infrastructure, including the primary and secondary stacks, is defined as code using tools like Terraform or Ansible, enabling automated and consistent deployments.

4.4 Implementation Guidelines
- All service implementations should be designed to be loosely coupled and easily replaceable
- Configuration-based switching should be implemented as a cross-cutting concern, affecting all layers of the system
- Infrastructure as code should be used to manage the deployment and provisioning of the entire technology stack
- Comprehensive testing, including integration and end-to-end tests, should be in place to validate the switching mechanism

5. Communication Patterns
5.1 Synchronous Communication
- Synchronous communication is handled through the service mesh layer, which provides service discovery, load balancing, and circuit breaking
- The API gateway and middleware components act as the entry points for synchronous client requests, routing them to the appropriate microservices

5.2 Asynchronous Communication
- Asynchronous communication is facilitated through the event infrastructure layer, which uses the message bus (Google Cloud Pub/Sub or Apache Kafka) for reliable event publishing and consumption
- Event processors, implemented as serverless functions or managed services, handle the asynchronous business logic

5.3 Event-Driven Patterns
- The HexProperty backend follows an event-driven architecture, where microservices publish and subscribe to domain events
- The event infrastructure layer ensures the reliable delivery and processing of these events,

enabling loose coupling and scalability

5.4 Service-to-Service Communication
- All service-to-service communication, both synchronous and asynchronous, is handled through the service mesh layer
- The service mesh provides features like service discovery, load balancing, circuit breaking, and secure communication, ensuring the overall reliability and resilience of the system

6. Scalability and Performance
6.1 Scaling Strategies
- Horizontal scaling of microservices using auto-scaling policies in the compute layer (e.g., Cloud Run's autoscaling)
- Vertical scaling by adjusting resource allocations (CPU, memory) based on usage patterns
- Scaling of the event infrastructure and data stores based on throughput and storage requirements

6.2 Performance Considerations
- Caching and query optimization techniques for read-heavy workloads
- Asynchronous processing and event-driven architectures for decoupling and scalability
- Distributed tracing and performance monitoring to identify bottlenecks

6.3 Resource Management
- Efficient resource utilization through the use of managed services and serverless offerings
- Dynamic scaling of resources based on demand patterns
- Resource isolation and multi-tenancy support

6.4 Load Handling
- Load balancing and circuit breaking at the service mesh layer to handle sudden spikes in traffic
- Queueing and batching mechanisms in the event infrastructure layer to smooth out load fluctuations
- Autoscaling policies to dynamically adjust resource allocations based on load patterns

7. System Boundaries
7.1 External Interfaces
- API Gateway as the single entry point for client applications and external integrations
- Pub/Sub or Kafka as the interface for event-based integrations with other systems

7.2 Internal Boundaries
- Clear boundaries between microservices, enforced through the service mesh
- Separate data stores and caching mechanisms for read and write workloads

7.3 Integration Points
- Secure communication channels between microservices using the service mesh
- Event-driven integration between microservices through the event infrastructure layer

7.4 Security Boundaries
- Authentication and authorization handled at the API Gateway and service mesh layers
- Secure communication between microservices using mTLS
- Secure access to data stores and caching mechanisms

8. Configuration Management
8.1 Environment Configuration
- Environment-specific settings, such as connection details, resource limits, and feature flags, are stored in a centralized configuration repository
- Configuration changes can be applied at runtime without redeploying the entire system

8.2 Feature Flags
- Feature flags are used to enable or disable specific functionality in a controlled manner
- Feature flags can be used to gradually roll out new features, perform A/B testing, or quickly disable problematic functionality

8.3 Infrastructure Configuration
- The entire infrastructure, including the primary and secondary technology stacks, is defined as code using tools like Terraform or Ansible

- This enables automated and consistent deployments, as well as the ability to quickly switch between the primary and secondary stacks

8.4 Stack Switching Mechanism
- The configuration-based switching mechanism is implemented as a cross-cutting concern, affecting all layers of the system
- The switch is achieved by updating the centralized configuration repository, which triggers the necessary changes in service registrations, dependency injections, and infrastructure deployments

9. Development Workflow
9.1 Code Organization
- The codebase is organized into independent, loosely coupled modules that align with the microservices architecture
- Each microservice has its own domain, application, infrastructure, and interface layers

9.2 Dependency Management
- Dependencies are managed using tools like Poetry or pip-tools, ensuring consistent and reproducible environments
- Cross-service dependencies are managed through the service mesh, ensuring loose coupling and easy replacement of implementations

9.3 Build Process
- The build process is automated and triggered by changes to the codebase
- Containerized builds and deployment artifacts are produced, ensuring consistency across environments

9.4 Deployment Pipeline
- The deployment pipeline is fully automated, using infrastructure as code to provision and configure the necessary resources
- The pipeline supports both the primary (GCP-based) and secondary (vendor-independent) technology stacks, allowing for seamless switching between them

10. Future Considerations
10.1 Evolution Strategy
- The architecture is designed to be flexible and adaptable, allowing for the gradual introduction of new technologies and patterns
- The configuration-based switching mechanism enables the exploration and adoption of new technology options without disrupting the overall system

10.2 Technology Roadmap
- Periodically evaluate the primary and secondary technology stacks to ensure they remain competitive and aligned with the system's evolving requirements
- Continuously monitor the ecosystem for emerging technologies and trends that could benefit the HexProperty backend

10.3 Scalability Planning
- Regularly assess the system's scalability needs, both in terms of data volume and traffic, to proactively plan for infrastructure upgrades and scaling strategies

10.4 Maintenance Strategy
- Implement a comprehensive observability and monitoring solution to quickly identify and address performance issues or service degradations
- Establish a robust incident response and disaster recovery plan to ensure the system's resilience and availability

</document_content>
</document>