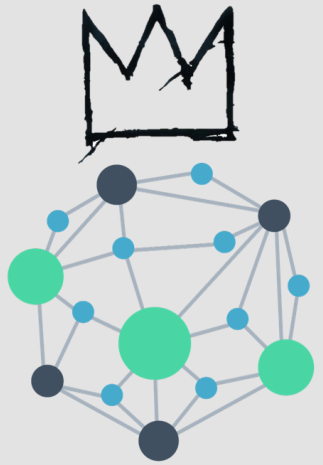
KMAP
Technical overview



Knowledge is King

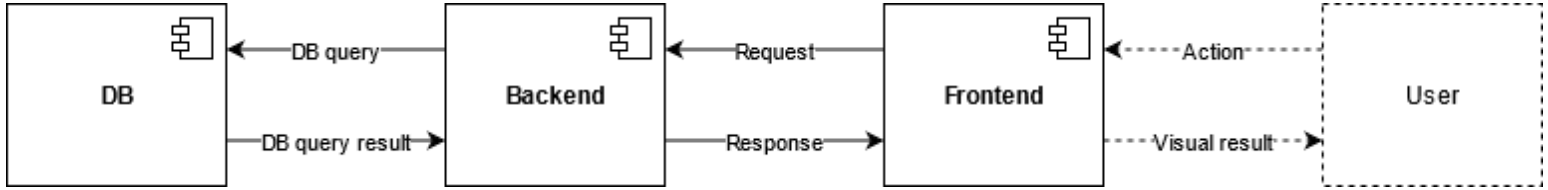
Johannes Jablonski
Tobias Kopyto
Igor Shmelev (ret.)
Cato Trütschel
Jule van de Logt
Thomas Wehr
Yannick Zuber

KMAP – Technical overview

- High-level component overview
- Database
- Backend
- Frontend
- Query archetypes

High-level component overview

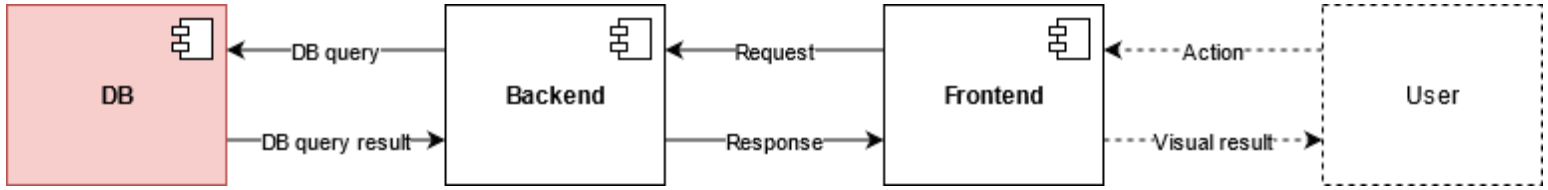
High-level component overview



- Three-tire architecture
  - Database
  - Backend (incl. Data-access layer)
  - Frontend
- Database is not publically accessible
- Database can stay (logically) external
  - Use existing databases
  - No publically available HTTP end-point
- Backend can pre-process request and post-process responses

Database

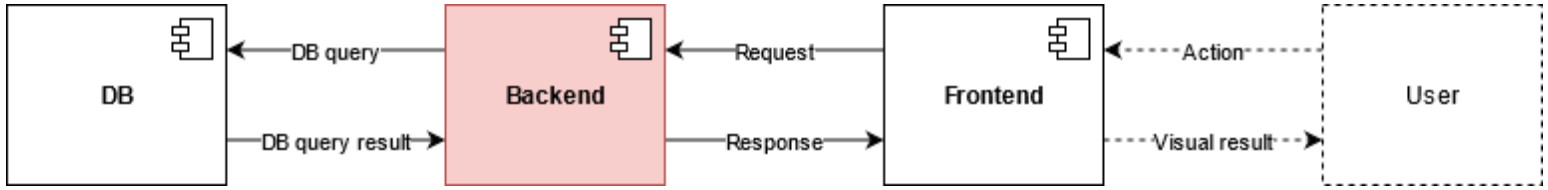
Database



- Requirements
  - Existing or new graph database
  - No write access required
  - Schema and data should not change currently due to caching (WIP)
  - DB system is pluggable by replacing DAL in backend
- Currently used DBMS is neo4j
  - Minimal configuration
  - APOC and GDS plugins needed (fallbacks are WIP)

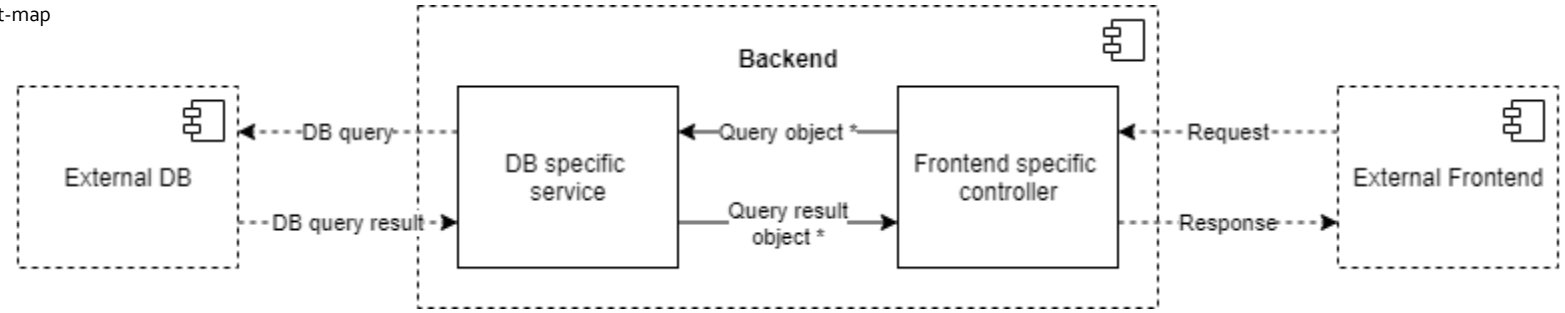
Backend

Backend



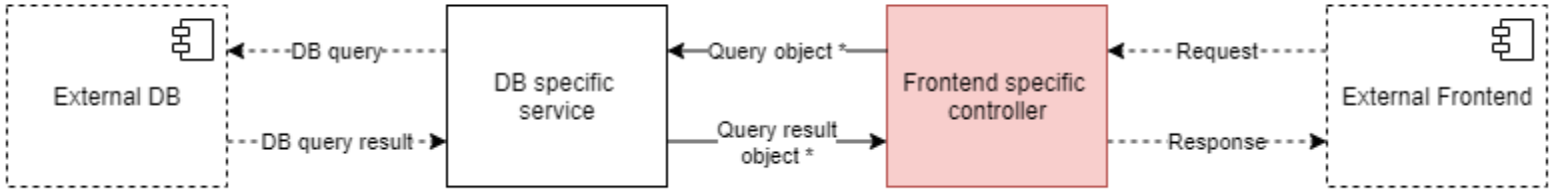
- Provide domain specific data-service
  - HTTP API via a webserver for web-clients (frontend)
  - Process domain specific queries from frontend
  - Formulate DB-Queries, execute them via the DB instance
  - Post-processe query results
- Additional application-wide services
  - Data-indexing
  - Caching
  - Data-projection
- Technologies and frameworks
  - Node.js https://nodejs.org/en/
  - NestJS https://nestjs.com/

Backend – Components



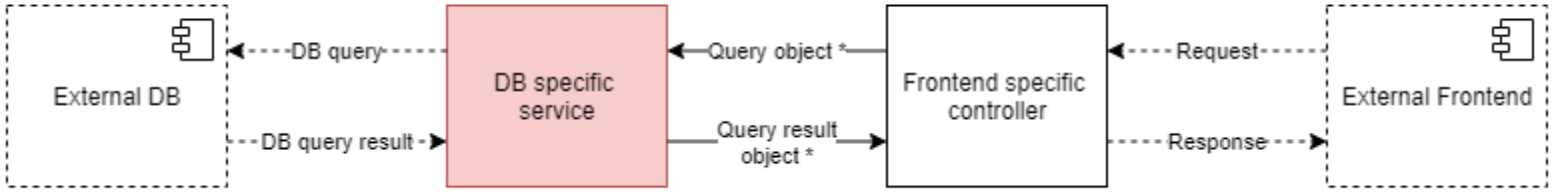
- Internal *Backend* components
  - DB specific services
  - Frontend specific controllers
- Reason for split
  - Adaption to two separate and distinct actors (SRP)
  - Support of testability via component tests
- Domain-specific *query* objects and *query-result* objects

Backend – Controller



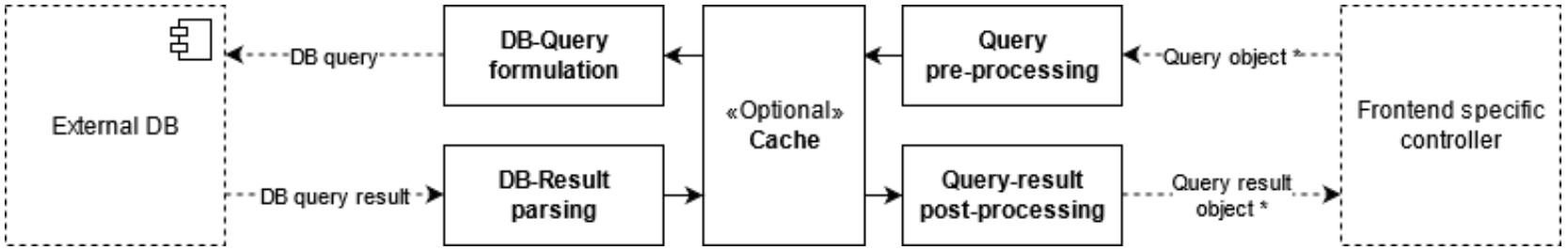
- Responsiblity
  - Error-handling and validation of requests
  - Construction and augmentation of *request* objects
  - Ensure backwards-compatibility and support future API changes (via translation)
- NestJS controllers
- Makes available HTTP API to web-clients (frontend)
- API adapted to and specialized for frontend needs

Backend – Service (1)



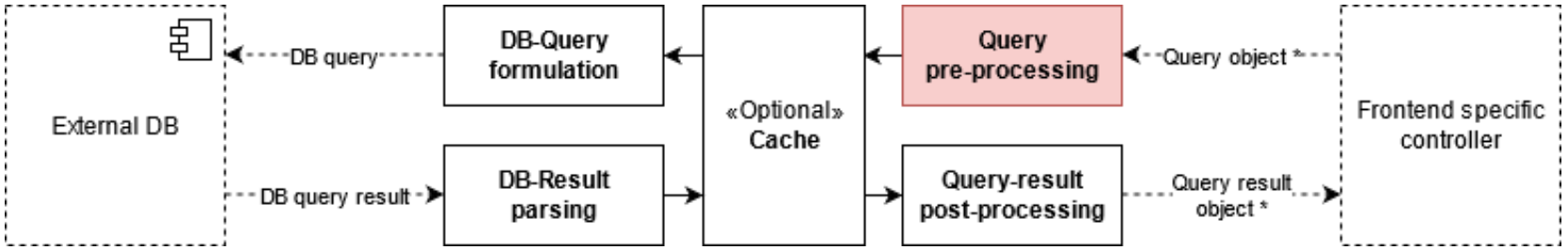
- Responsiblity
  - Process domain specific request (*query* objects) and deliver self-contained response (*query-result* object)
- Injected, composable services
- Adapted to and specialized for the used database
- Includes the data-access layer (DAL)
- Support for other DBMS then neo4j
  - DB dedicated implementation of service interface for target DBMS
  - Replacement of injected service in service composition

Backend – Service (2)



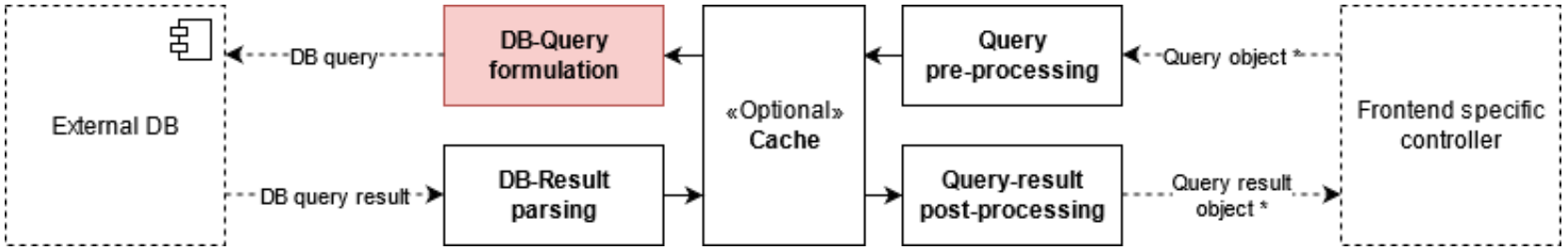
- Detailed structural design of a backend service
- Dataflow
  - Query object preprocessing
  - DB-Query formulation
  - DB-Query execution (external)
  - DB-Result parsing
  - Query-result post-processing
- Optional caching
- Other specialized services

Backend – Query pre-processing



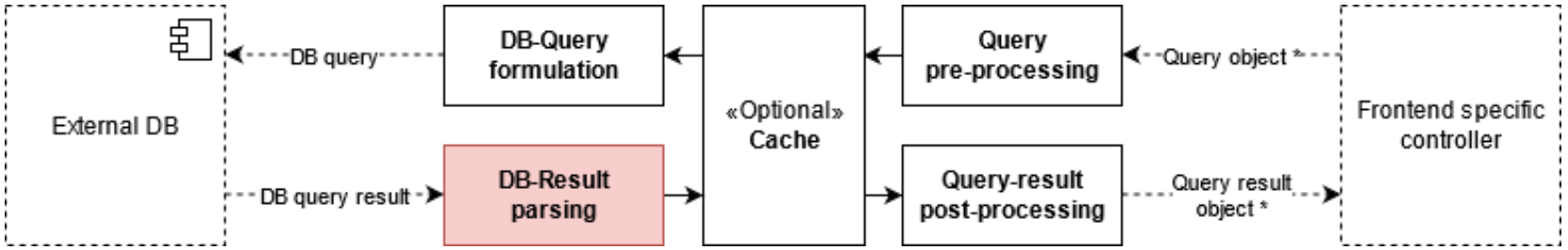
- Pre-processing of *query* objects
- Translation into internal (cacheable) format
- Preparation of query-formulation
- Example
  - Split into separate requests of nodes and edges
  - Split into two subqueries that are executed and the results combined

Backend – DB-Query formulation



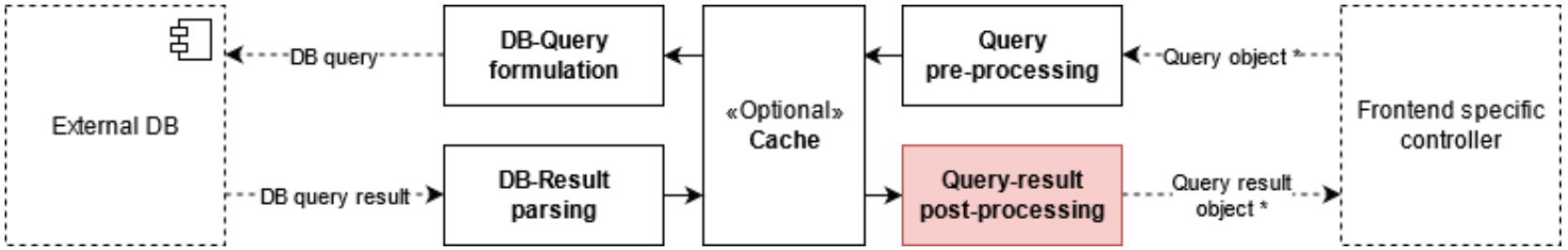
- Formulation of DB-query
- Using the DB specific query language (cypher for neo4j)
- Setting parameters from the *query* objects (or its internal representation)
- Construction of (rather) dynamic queries
  - Translation of domain specific query-language
  - Deep-first tree traversal
  - Used for filter-queries (see *query archetypes*)

Backend – DB-Result parsing



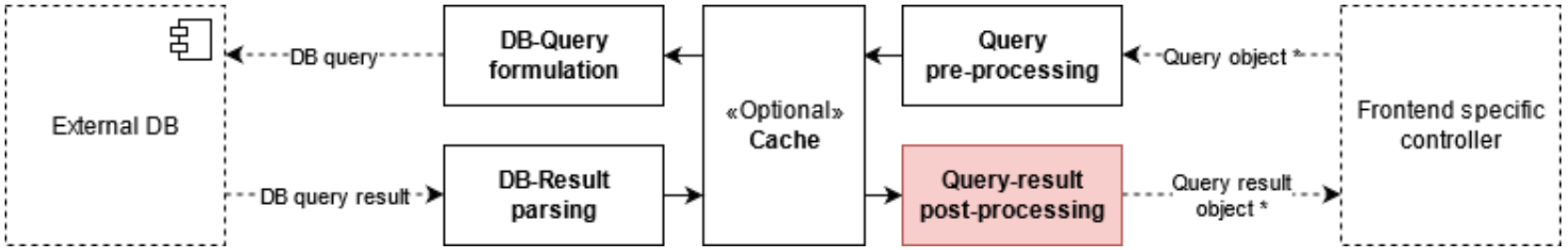
- Translation of DB query results to *query-result* objects (or its internal format)
- DB-Errors processing
  - Failure
  - No result

Backend – Query-Result post-processing (1)



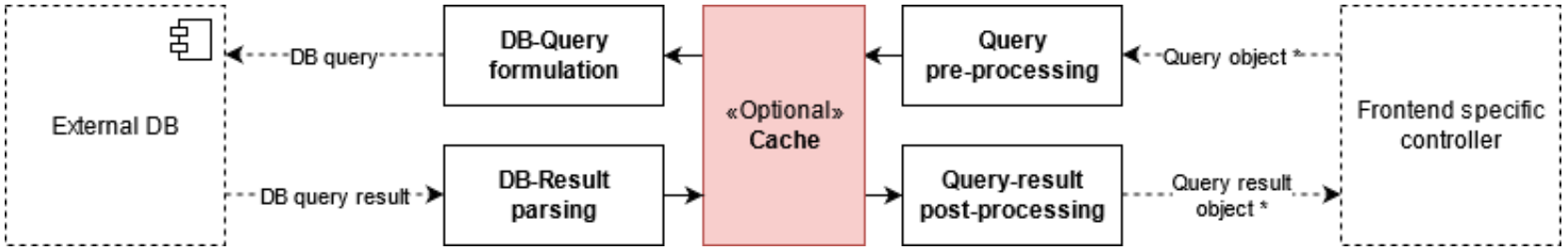
- Post-processing of *query-result* objects
- Translation from the internal format (if required)
- Combination of results
  - If additional services were used for execution
  - If query pre-processing split request into multiple parts
- Query-result object consolidation
  - Removing duplicate entries
  - Handle subsidiary nodes

Backend – Query-Result post-processing (2)



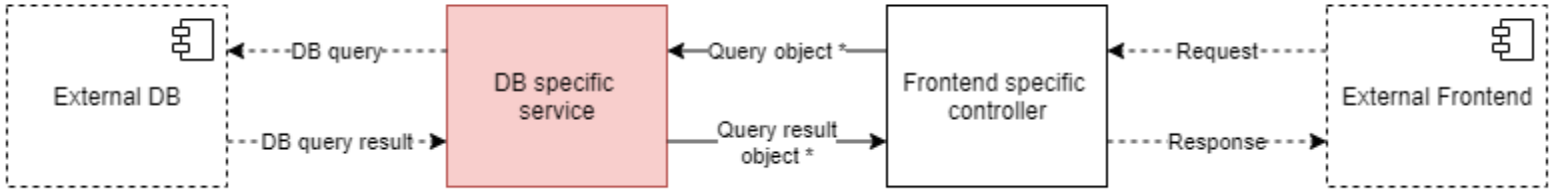
- Adding values from cache if present
- Example
  - Combine edge and node query result to a self-container query-result
  - Combine filter-result as received from external service with special query (See shortest-path query)

Backend – Optional Caching



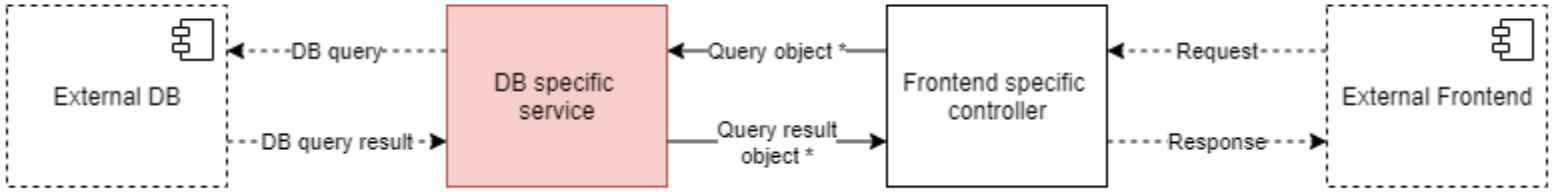
- Optional query-result caching
- Cache complete query-result for given key
- Cache partial query-results
  - Query pre-processing removes all query arguments that cache contains result for
  - Query-result post-processing combines cached results with results from databat
- Cache is currently not updates and assumes a read-only database (WIP)
- Example
  - Caching of data schema
  - Search-index

Backend – Specialized services



- Specialized domain-specific services
- Cannot directly be represented with the service-pipeline
- Services are application global
- Examples
  - Projection of data-set into domain specific format for futher processing
  - Indexing the dataset for fast full-text lookup

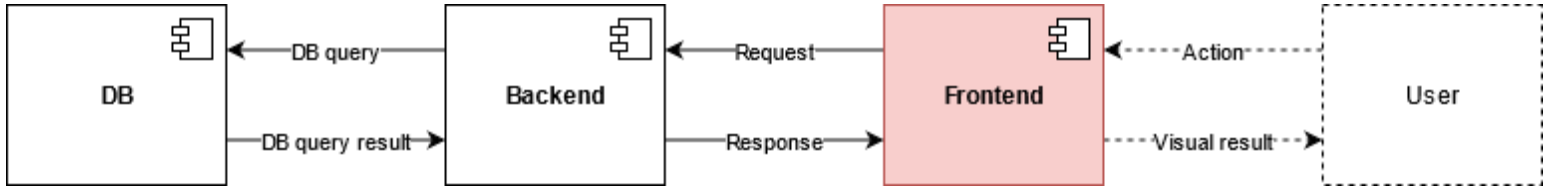
Backend – Composability



- Backend services are composable
- Services are injected via DI system
- Can built up on each others functionality
- Example:
  - Shortest-path service uses filter-service and combines results
  - Search-service uses general query-service to build search-index

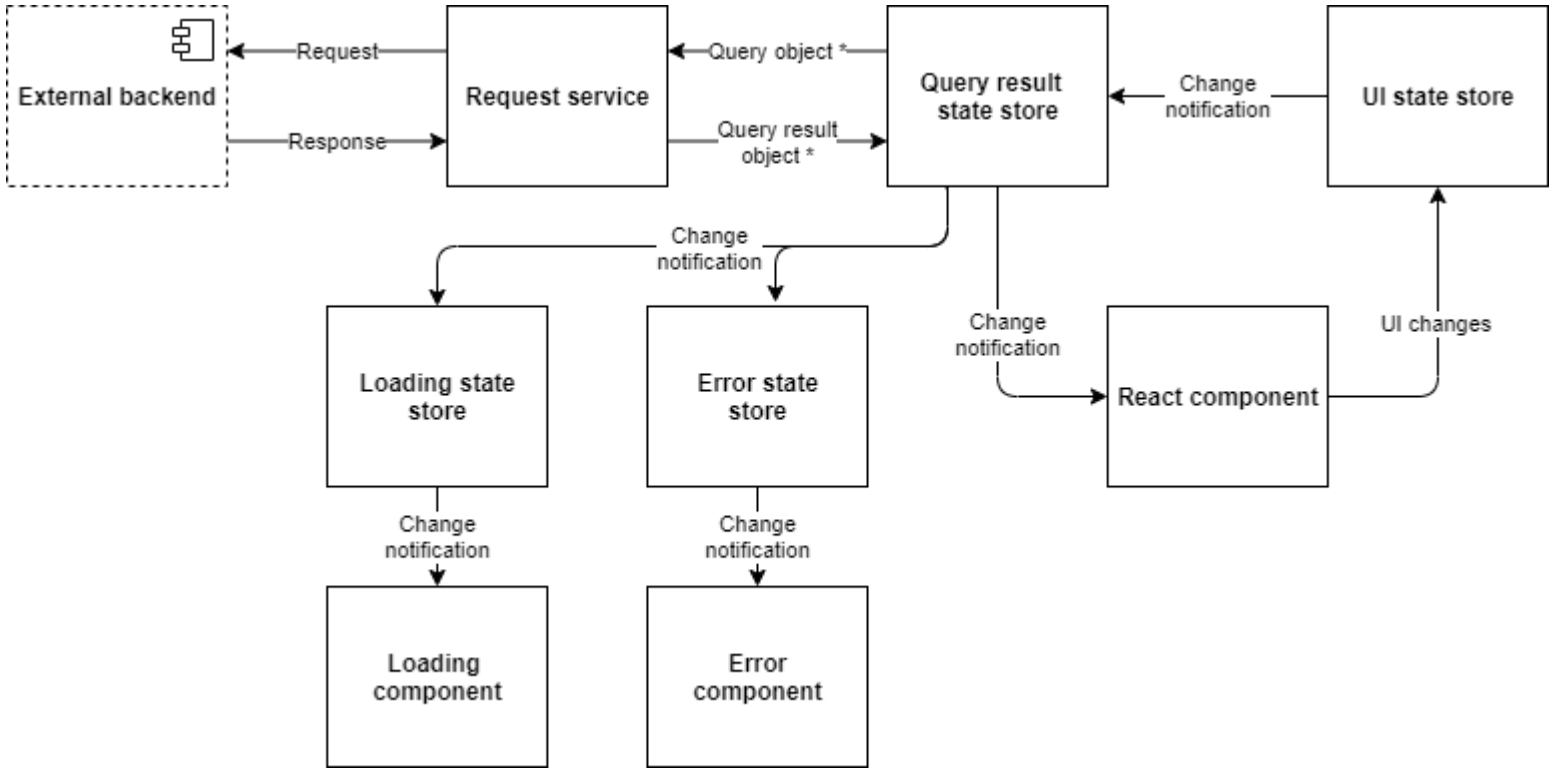
Frontend

Frontend

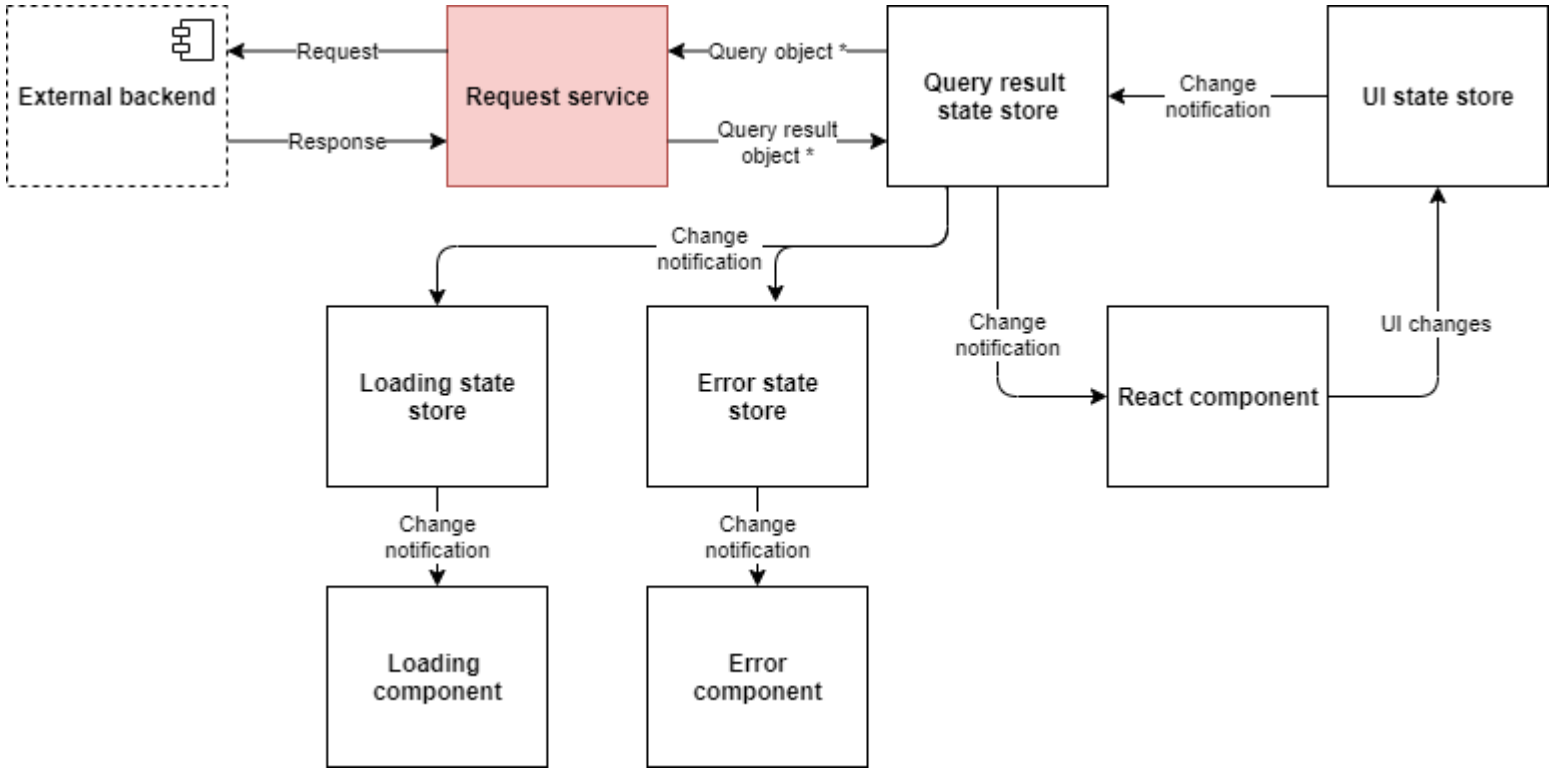


- Single page application (SPA) in the user's browser
- Displays graph and exploration results to user
- Processes action (request, queries, explorations) via HID
- Technologies and frameworks
  - Modern browser (Firefox, Chrome, Edge, Safari, etc.)
  - React https://reactjs.org/
  - Inversify https://inversify.io/
  - Material UI https://material-ui.com/
  - RxJS https://rxjs.dev/

Frontend – Components



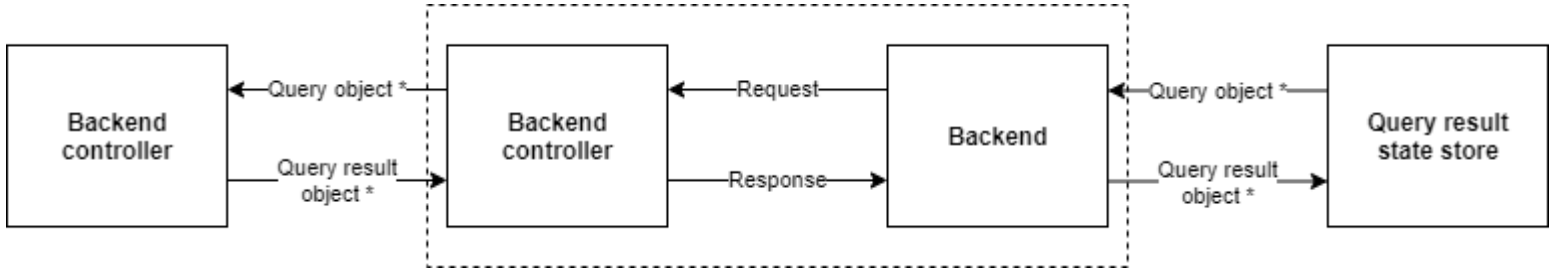
Frontend – Request service (1)



Frontend – Request service (2)

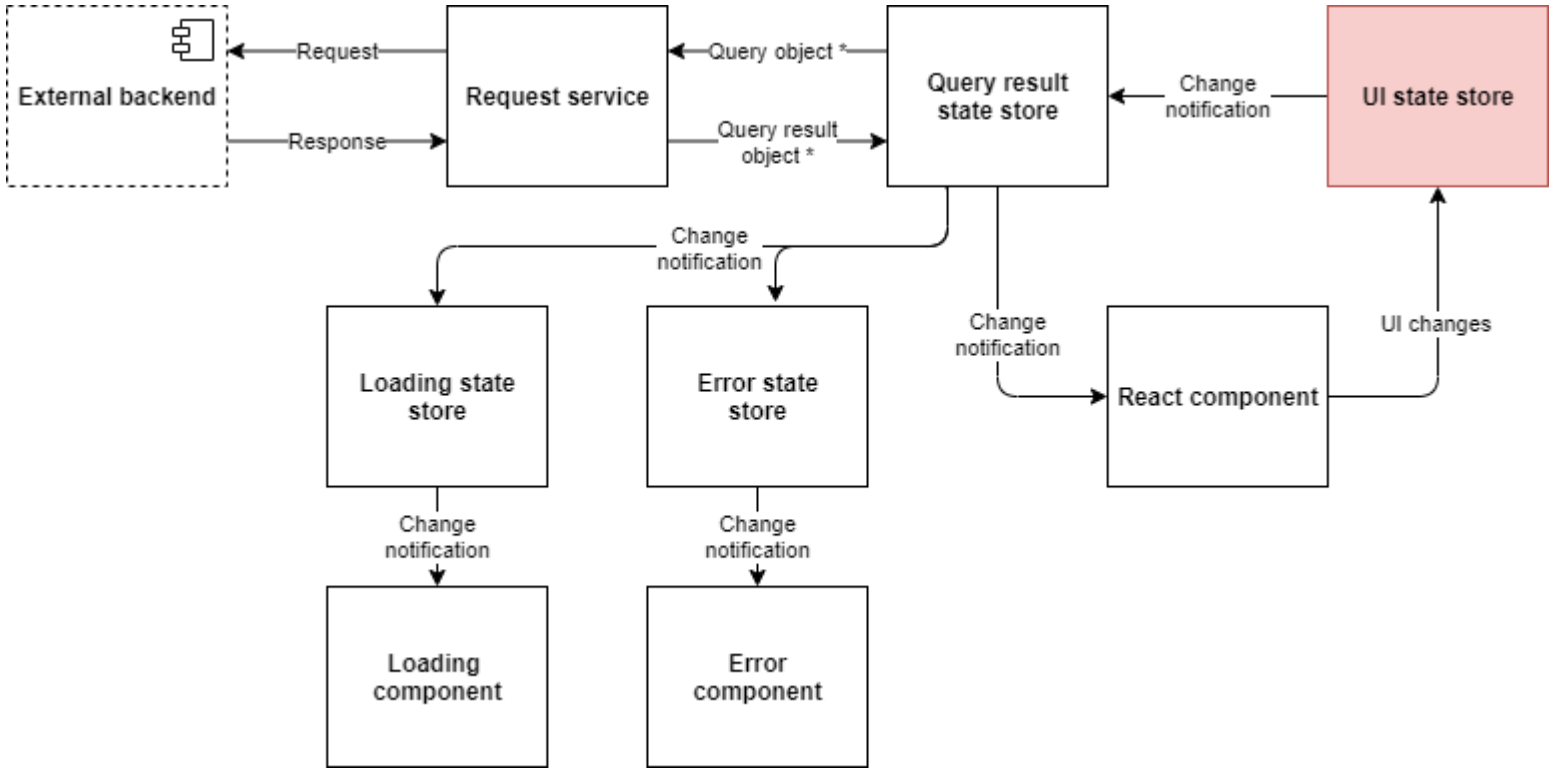
- Responsible for communiction with backend
- Translates *query* objects from the app into HTTP calls
- Translates the HTTP Response from the backend into *query-result* objects
- Uses JSON serialization
- Error handling
  - Network errors
  - Bad requests
  - Internal server errors
- Uses asynchronous IO via Promises (no blocking IO)

Frontend – Request service (3)



- Together with the respective backend controller
  - Domain specific asychronous RPC infastructure
  - *Query* objects are passed through to the backend
  - *Query-result* objects are passed back to the frontend

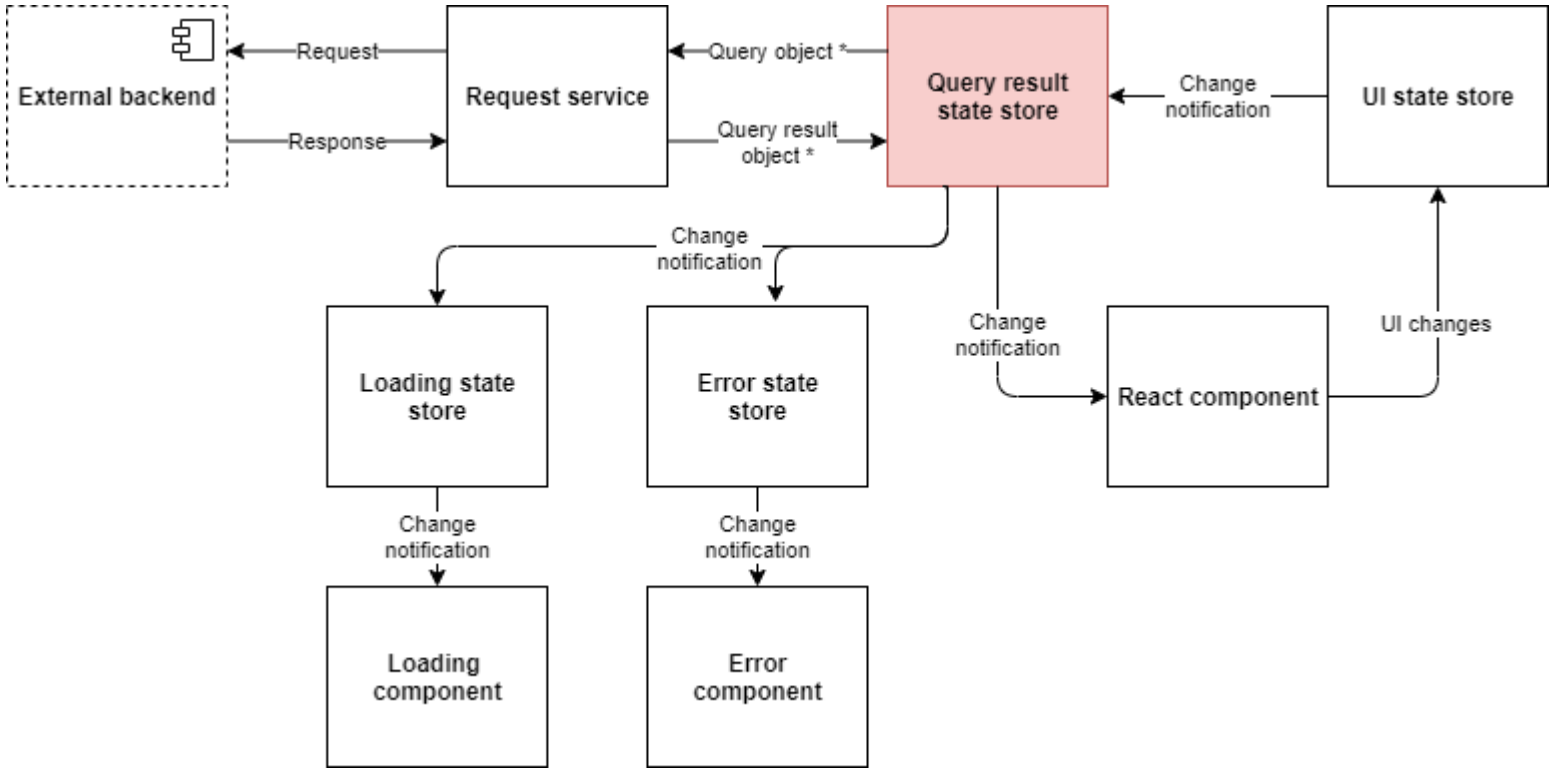
Frontend – UI state store (1)



Frontend – UI state store (2)

- Stores the current state of the user-interface (UI)
- Decoupled state from the presentation of the state
- Modularization of application
- Notifies other components (that are subscribed) about UI state changes
- UI state is preserved when displayed page is switched
- Future use: Store state in local-storage for persistence

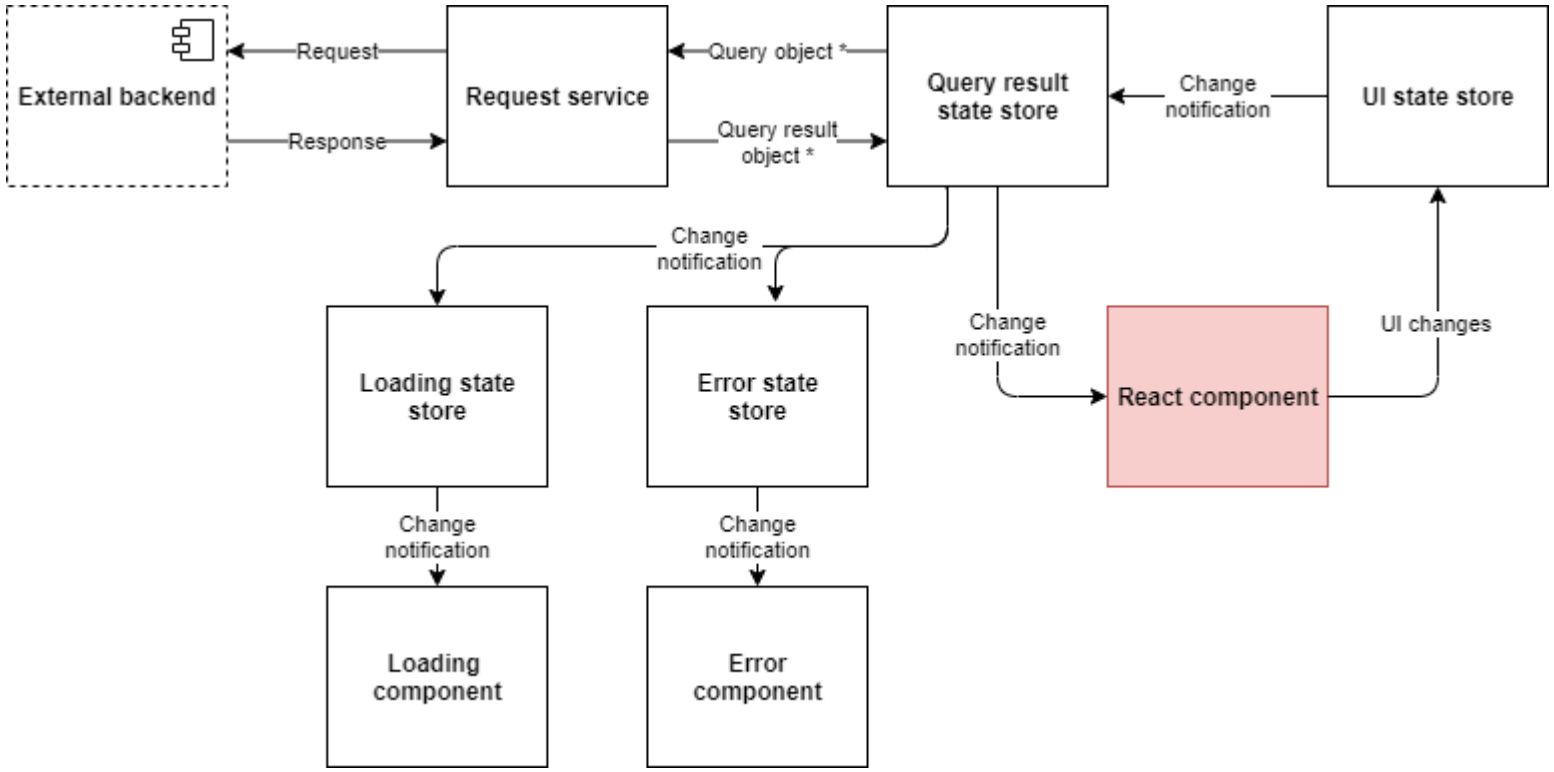
Frontend – Query result state store (1)



Frontend – Query result state store (2)

- Stores the result of the latest query-operation
- Updates the state when the *UI state-store* notifies about changes
- Builds query from the UI state
- Executes query via the *request-service*
- Notifies UI components about changes of state
- Handles also
  - Cancellation
  - Throttling
  - Loading
  - Errors

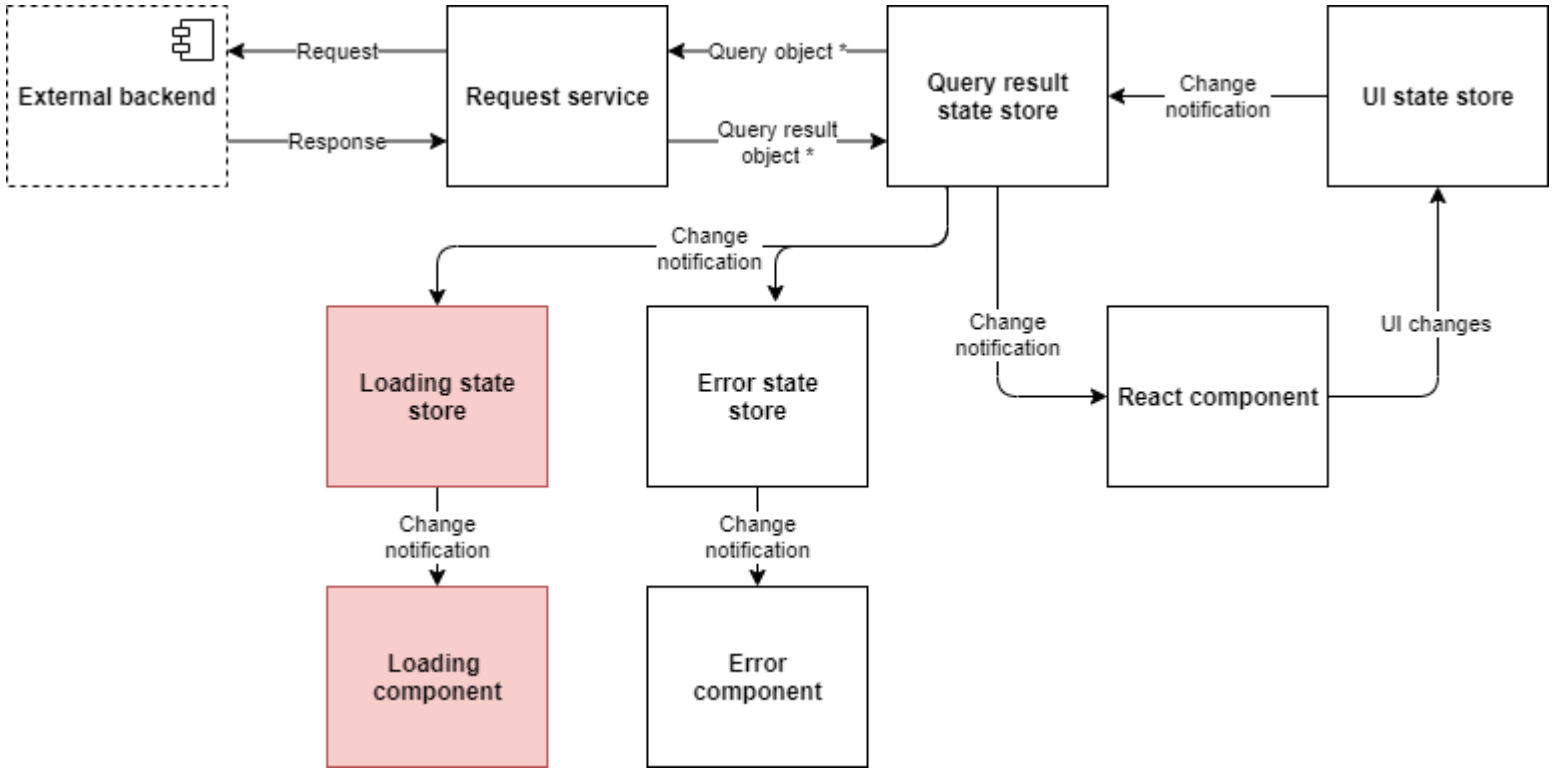
Frontend – UI Component (1)



Frontend – UI Component (2)

- Display the state, as stored in the subscribed *UI state-store*
- Can also display state in secondary stores (like the *query-result state store*)
- When not a read-only view, allow state changes via the UI
- Update the *UI state-store* about updates in the UI
- *UI components* are implemented via React components

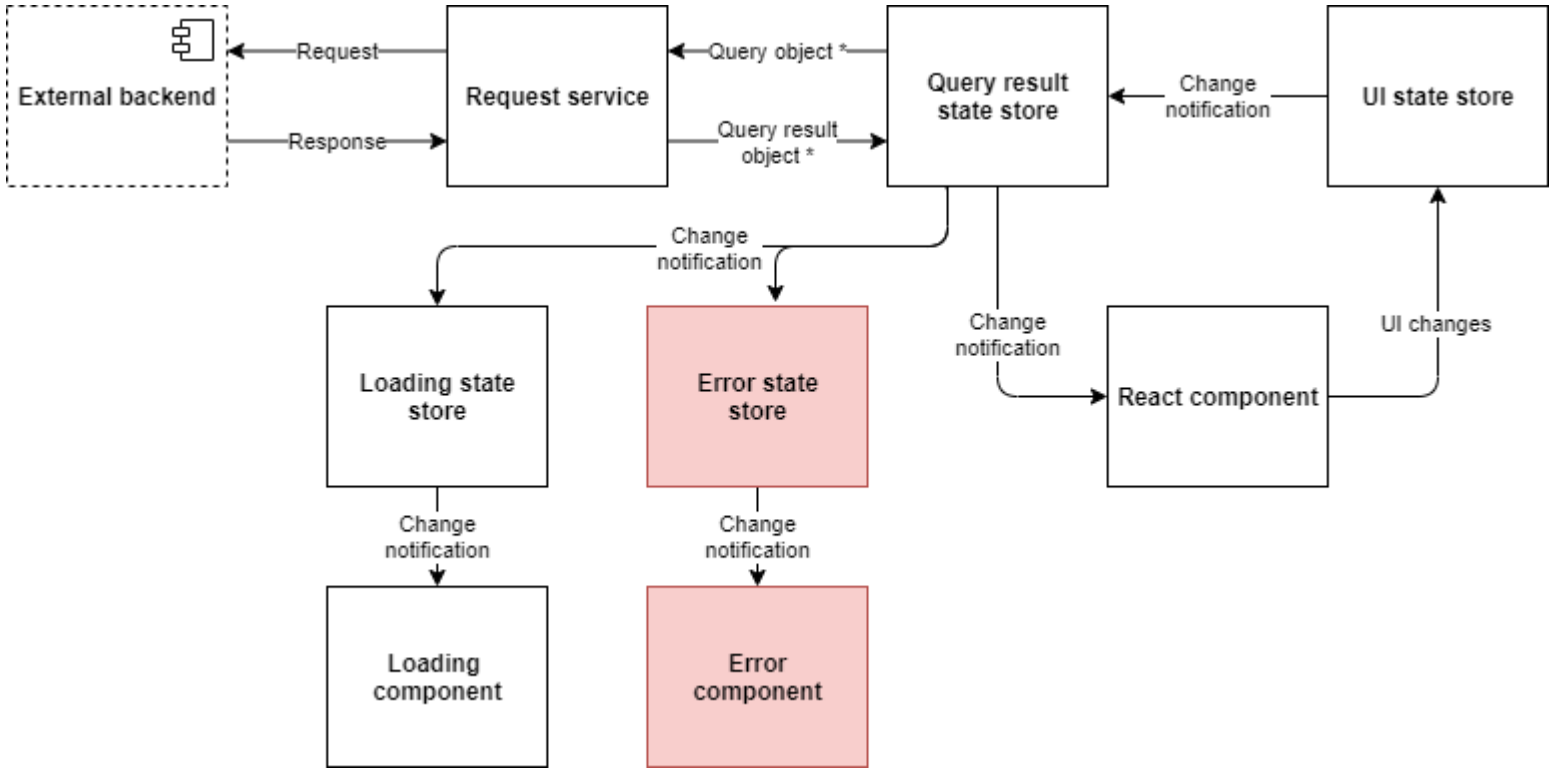
Frontend – Loading (1)



Frontend – Loading (2)

- A dedicated state store
- A dedicated *UI component*
- Store and display the loading operations in the current system
- Are notified of loading operations by other stores and services
- Display an option for cancellation that a loading operation can be canceled with

Frontend – Error (1)



Frontend – Error (2)

- Dedicated state store
- Dedicated UI component
- Store and display the latest application error
- Error types
  - Cancelation errors
  - Network errors
  - Server errors
  - Page not found (HTTP 404) errors
  - etc.
- Notified by other services, state stores and the page router about errors

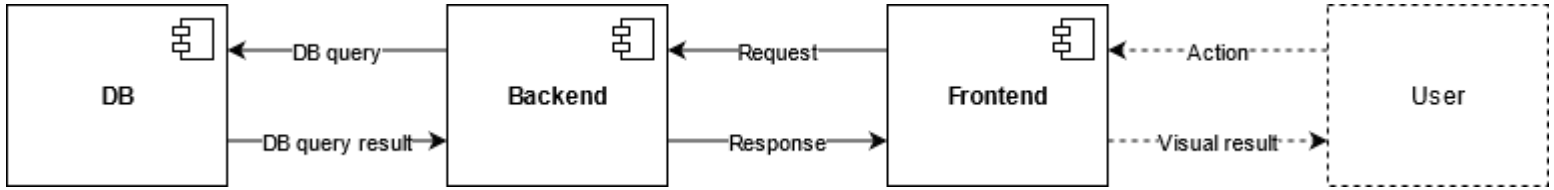
Query archetypes

Query archetypes

- Graph query
  - Filter
  - Shortest-path
  - etc.
- Schema query (without further insight)
- Search query (without further insight)

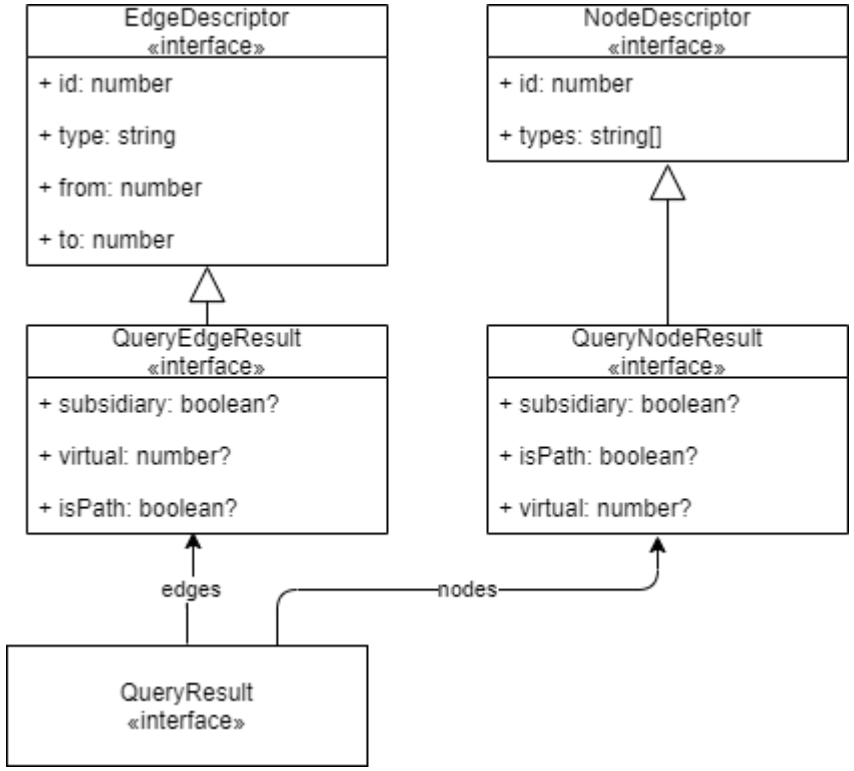
Graph query

Graph query



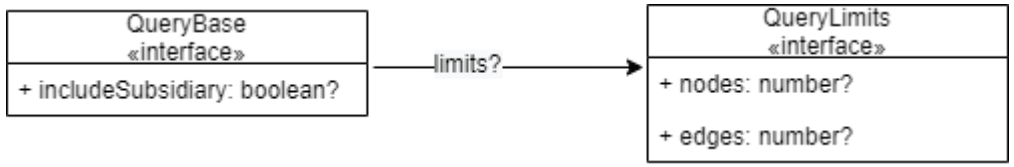
- Basic structure of all graph-queries
- User specifies meta-parameter via action
  - Filter settings
  - Start and end-node of shortest-path search
  - Etc.
- Domain specific *query* object is constructed in the frontend
- Backend processes *query* object and sends back *query-result* object
- *Query-result* object is displayed in the fronend

Graph query – Query result object



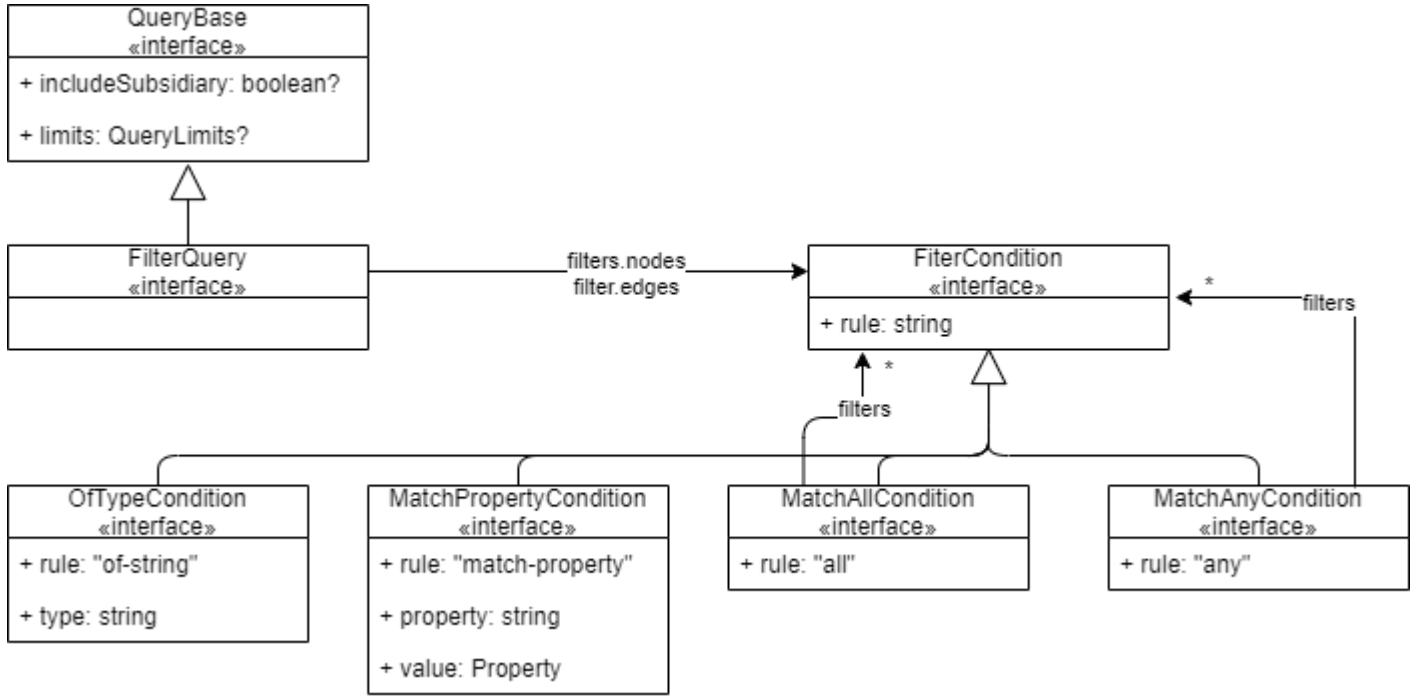
- Common *QueryResult* type
- Visualizations do not need to be adapted to query-archetype (Mix&Match)
- Entity properties
  - Subsidiary
  - Virtual
  - IsPath
- Query result is guaranteed to be consistent (See backend query-result post-processing)

Graph query – Query object



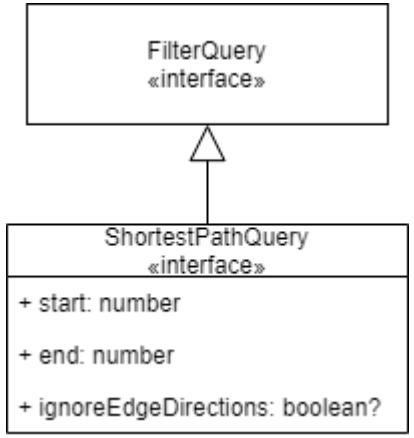
- QueryBase interface as a common subtype for all queries
- Subsidiary filter option
- Node and edge limit

Graph query – Filter query



- Formulation of arbitrary filters (predicates)
- Extensible to additional predicates
- Translated to DB specific query via deep search tree-traversal

Graph query – Shortest path query



- Extended *FilterQuery*

Graph query – Entity details

- *Query result* objects only contains most relevant information about edges
- How to access entity details (i.e. property values)
- Solution: Specialized query-service
- Cache by the entity id of entity details in the frontend

Schema query

Schema query

- Allows to query the schema
- Node types present in the dataset and their properties
- Edge types present in the dataset and their properties
- Which edge-type connects which node types (WIP)
- Massive caching in the frontend

Search query

Search query

- The *query* object is a single string
- Possibility to enhance the query object in the future for further options
  - Search only in specific fields
  - Case-sensitivity
  - etc.
- Search result
  - Nodes
  - Edges
  - Node types
  - Edge types