Working with Appian's Primary Datasource

Estimated completion time: 1 hour

Course Outline

Introduction

This course is directed at Appian's Primary Data Source, where we store relevant Appian-specific data (Record Types, News Entry Data, etc.)

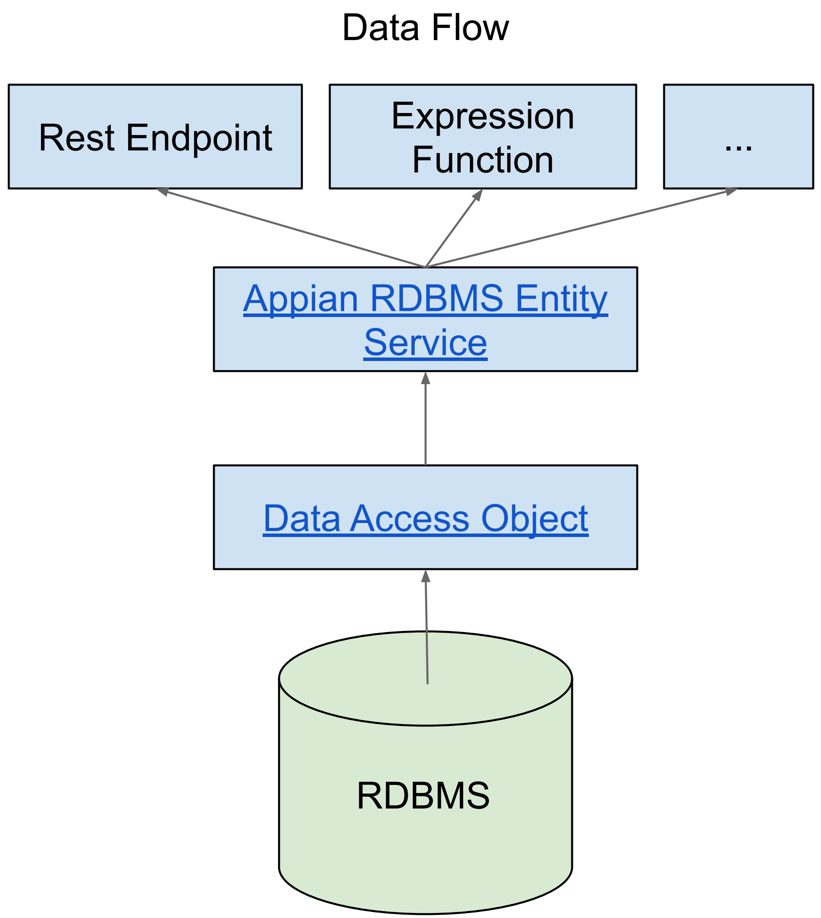
RDBMS is a term that we use a lot at Appian. It stands for **R**elational **D**ata**B**ase **M**anagement **S**ystem. It is what allows us to access, and populate data within a relational data source.

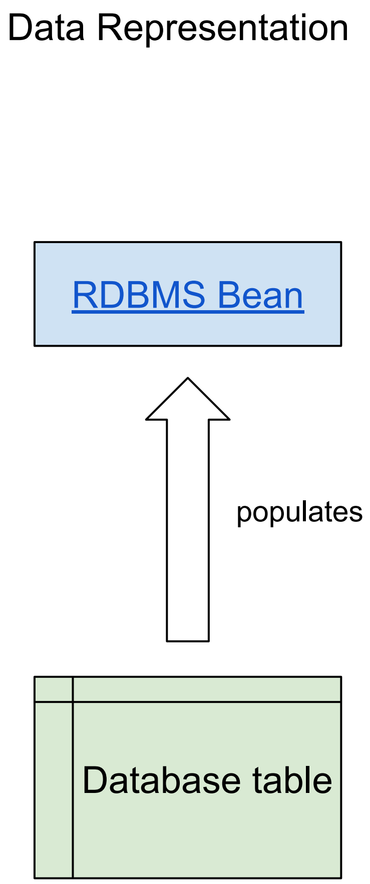
Appian data is persisted to K or the Primary DS. K is an in-memory database that we use for a lot of our real-time data like process execution, or data that does not contain complex relationships to other data. Relational databases are an easy way to represent these relationships, and therefore we have begun using RDBMS more frequently.

A customer can run appian on top of MariaDB, MySQL, Oracle, DB2, SQLServer or Postgresql. Hibernate is a database technology that allows us to store our data across any of these types.

This course will walk you through the basic code structure of how we work with and create data for use within Appian's Primary DS. Upon finishing this course, you should be able to quickly create an RDBMS backend for any feature and understand the ones that already exist.

Overview of Components





Data Access Object (DAO)

What is a DAO?

- Data Access Object - is an object that provides an abstract interface to some type of database or other persistence mechanism.
- Every Appian DB object has a DAO
- Using a DAO means the developer doesn't need to know how to communicate with Oracle, MySQL, etc. They just need one interface.
    - Hibernate offers APIs that let you make queries and specify criteria. Some commonly used classes are (if you're interested):
        - Criteria (which rows you're looking for)
            - Restrictions (how you build Criteria)
        - Projections (what columns in those rows you care about)

Making a DAO DEPRECATED

- Every DAO is an interface/implementation pair.
  - Naming convention is: [Object]Dao and [Object]DaoHbImpl
  - Hb stands for Hibernate, since that is our library of choice for database interaction.
- The interface must extend `GenericDao` or, if your object requires a RoleMap, `GenericDaoWithRoleMap`
  - *Note:* A *RoleMap is how we configure security on Appian objects. A "Role" could be things like Editor, Viewer, Admin. This way we can restrict different actions on a per-User or per-Group basis.*
- The DAO and the Java Bean we will learn about later are registered in `DataConfiguration.java`. This is the link between Hibernate and the DAO/Entity java classes that we make when setting up our new object.

Making a DAO (new)

This page is a WIP

In the meantime, see https://docs.appian-stratus.io/tech-docs/components/rdbms.html#dao

A Mouthful on DAO Views

Some of our more complex relational objects have a `View` class implemented on their DAO.

Views allow us to only pull back data we need at the service level (specifically any relationship annotated in the Java bean with `FetchType.LAZY`).

- For example, to display a record type list in tempo, we only need its name and some simple metadata, not all of its related actions, detail view configs, etc.

You may see a private initialize() method in some of our DAOs. That method is responsible for initializing *only* the data you want.

Testing Your DAO

Write **unit** tests for any conditional logic in the DAO, but always write **integration** tests at the service level - see RDBMS SOP.

See Cross-database Testing with TestContainers

RDBMS Beans

RDBMS Beans are Java objects that map to a table in RDBMS. They are defined using Java Persistence Annotations (JPA). At Appian, we use Hibernate as an implementation of the JPA specification.

Common interfaces implemented on these beans are `Id<Long>`, `Uuid<String>`, `Name`, `HasAuditInfo`, `HasRoleMap`. These interfaces will automatically hook up some common fields across all RDBMS objects (e.g. id, uuid, createdBy, updatedBy etc.)

Examples: SiteTemplate.java, WebApi.java

RDBMS Beans - Persistence Annotations

`@IgnoreJpa`

- This annotation is necessary for classes that map to CDTs
- It indicates that JPAs should be ignored when the class is parsed for CDT generation

`@Table`

`@Column`

`@OneToMany`

`@OneToOne`

`@BatchSize`

`@JoinColumn`

`@JoinTable`

`@PrePersist`

`@Transient`

@Table and @Column

These annotations are used to associate the database table and columns backing an entity with the bean for that entity. Here is an example table annotation for the SiteTemplate bean:

```
@Table(name = "site")
```
full code

And here is an example of annotating a getter on that same bean to associate the return value with a column

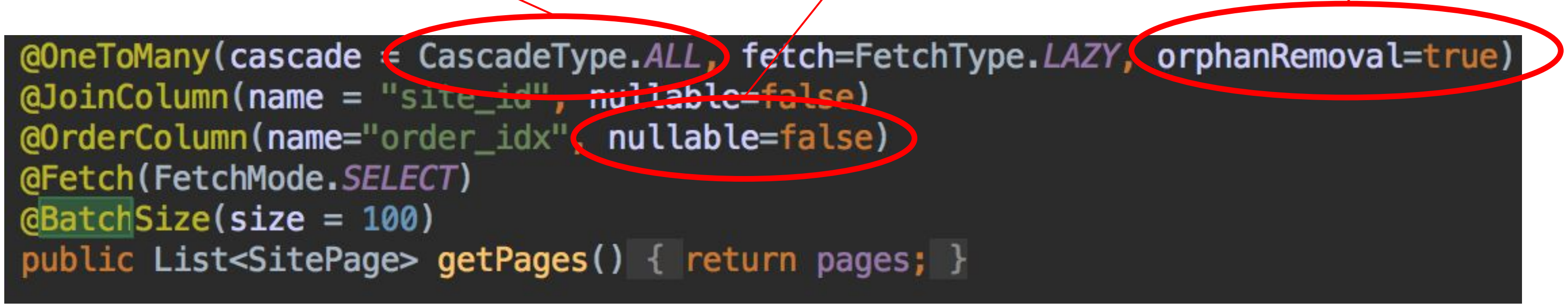
```
@Column(name = "description", length = Constants.COL_MAXLEN_MAX_NON_CLOB)
```
full code

Relationship Annotations

All CRUD operations will be performed on the child as well. You delete the Site, all of its pages will be removed as well.

Be explicit, even if you're not changing defaults

[ORM]() mapping for marking children for deletion when their parent is deleted.



```
@OneToMany(cascade = CascadeType.ALL, fetch=FetchType.LAZY, orphanRemoval=true)
@JoinColumn(name = "site_id", nullable=false)
@OrderColumn(name="order_idx", nullable=false)
@Fetch(FetchMode.SELECT)
@BatchSize(size = 100)
public List<SitePage> getPages() { return pages; }
```

The Dreaded LazyInitializationException

For performance reasons, we sometimes specify the `fetch = FetchType.LAZY` annotation on the relationship properties of our Hibernate entities. Some good explanations of LAZY vs. EAGER here can be found here and here.

A big problem with LAZY fetching of data is that **it only works when inside a Hibernate session (transaction)**. The lazy data object in the Java Hibernate entity object is actually a Hibernate *Proxy* that attempts to retrieve the real value from the database when it's accessed. This proxy will throw LazyInitializationException if the data was never fetched during the original transaction that created the entity object, but is then accessed later after the transaction has ended. This often[1] happens when a naive, non-`@Transactional` caller invokes a LAZY-annotated getter method on an entity!

Our service methods are almost always annotated with the `@Transactional` annotation, which is a Spring construct that's a little different than a Hibernate transaction (see here). Hibernate transactions for our purposes occur only within DAO code, and the transactions are generally committed (ended) before the method returns. **So using lazy collections on entities is really not an option** outside of the scope of a `@Transactional` method or the code called by such a method– otherwise this error occurs.

—

[1] But not always, so it's easy to get complacent

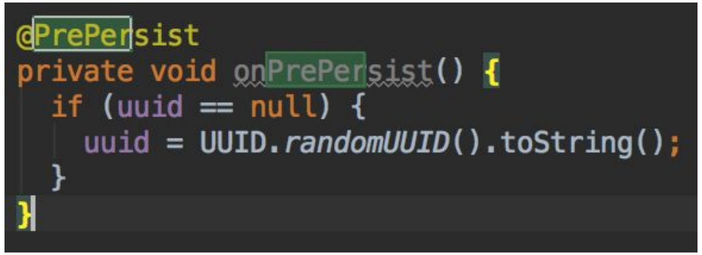
Avoiding LazyInitializationException

One way to avoid the LazyInitializationException is to call Hibernate.initialize() during the transaction. Our DAOs typically do this using Views - interfaces that tell us what to intialize, and restrict access in the application. This is confusing because developers need to know the interfaces exist and which one they need for their specific purpose. This practice is slowly being discontinued (see `RecordTypeDefinition.java`) as the requirements of laziness for performance concerns can be mitigated via distributed caches for these objects.

Another way to avoid LazyInitializationException is to ensure LAZY-annotated entity properties are only accessed by service methods, which are generally safened by their @Transactional annotations. It may be possible to apply Java **package** visibility (default access) to all the dangerous methods, and only call them via a bespoke service method; however, this only works if the service class is in the same package as the entity, and it doesn't protect against mistaken/unprotected calls from **non-service** classes in the same package.

@PrePersist and @Transient

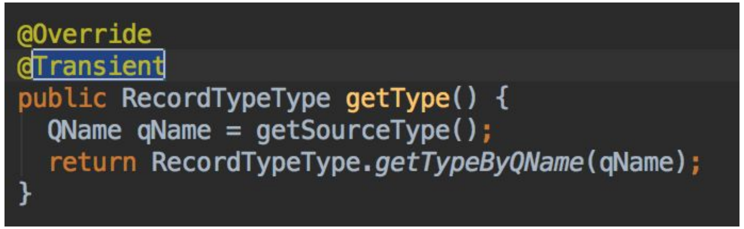
`@PrePersist` lets you perform certain actions right before we write to the DB.



```
@PrePersist
private void onPrePersist() {
  if (uuid == null) {
    uuid = UUID.randomUUID().toString();
  }
}
```

We want to auto-generate object UUIDs right before we persist.

`@Transient` marks a method so that hibernate ignores it. These are used for convenience methods when we want to perform a transform on bean fields.



```
@Override
@Transient
public RecordTypeType getType() {
  QName qName = getSourceType();
  return RecordTypeType.getTypeByQName(qName);
}
```

We only want to interact with the RecordTypeType enum, but we can't store that directly in the DB, so we create a transient method.

RDBMS Beans - IX Annotations

These annotations are only relevant for Appian objects that need to have the capability to be imported and exported.

`@ComplexForeignKey`

- Indicates that the bean property makes a reference to another Appian object

`@ForeignKeyCustomBinder`

- Specifies a binder to handle a complex foreign key
- Example: DocumentRefBinder associates document references with the appropriate Appian Document

RDBMS Beans - XML Annotations

These annotations are only relevant for RDBMS beans that need to be converted to XML

`@XmlType`

- Maps an RDBMS bean to its XML representation.

`@XmlTransient`

- Applied to getters for bean properties that should not be transferred to XML representation
- This is useful when the desired xml representation is a transformed version of the database representation

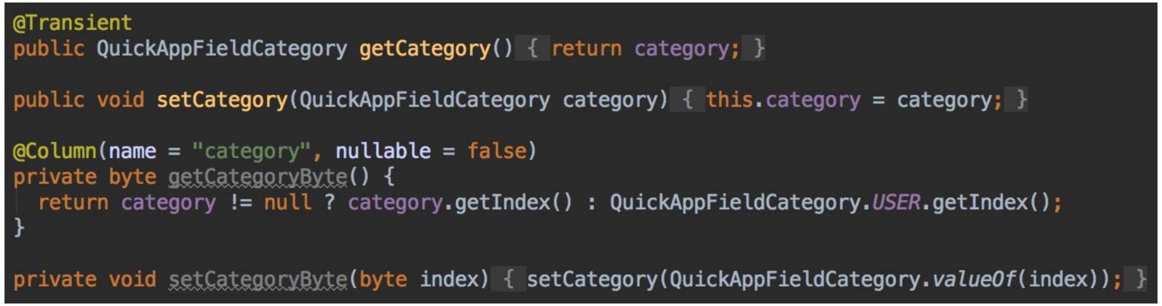
`@XmlElement`

- Applied to getters for bean properties that SHOULD be transferred to XML representation

RDBMS Beans - Fields based on Java Enums

If you want to create a column based off of an enum, we do this by storing a byte in the database which we use in the bean's get() to assign a value in Java. To do this:

- Create your enum with a name and a byte index.
- Create a private getter and setter for the byte on the RDBMS bean, on which the @Column annotation is placed.
- Create a private field for the enum type on the bean
- Create a public @Transient getter and setter on the bean
- Example from QuickAppField.java:



```
@Transient
public QuickAppFieldCategory getCategory() { return category; }

public void setCategory(QuickAppFieldCategory category) { this.category = category; }

@Column(name = "category", nullable = false)
private byte getCategoryByte() {
    return category != null ? category.getIndex() : QuickAppFieldCategory.USER.getIndex();
}

private void setCategoryByte(byte index) { setCategory(QuickAppFieldCategory.valueOf(index)); }
```

RDBMS Entity Services

Services are Java APIs used for CRUD of Appian RDBMS objects.

Example: SiteTemplateServiceImpl.java

RDBMS Entity Services - Class Structure

Each service should have an interface and implementation class.

The interface should extend EntityService and the implementor should extend EntityServiceTxImpl.

- If you encounter a service that does not follow this pattern, it's likely because the requirements of this service did not fit into what EntityService provides

RDBMS Entity Services - Annotations

Any method interacting with the object’s DAO should be prefixed with the `@Transactional` annotation. This ensures that a database transaction will either be successful or be rolled back safely.

- Note that only methods that are called from outside the service can be @Transactional - e.g. you can't have a `helper method that uses the annotation to start a transaction`

`@RequiresCurrentUserInPrimaryDataSource` will add the current user to the primary data source (RDBMS) if they are not currently stored there.

RDBMS Entity Services: Testing

Every method that you implement must be unit tested!

~~In addition, you ***must*** write tests for every method to record and check the number of SQL statements that are executed during the method call.~~

- ~~This ensures that our number of SQL statements is stable, performant and consistent over time.~~
- ~~*Bonus: It helps you understand what Hibernate is actually doing!*~~

For an example, search for the test corresponding to a particular RDBMS Entity Service in a class named *ServiceTest.java (e.g., SiteService.java ⇔ SiteServiceTest.java).

Liquibase: Overview

- Liquibase is a database migration tool that allows you to change the structure of database tables over time.
- The developer provides a list of migrations that need to occur via an ordered set of files called a changelog.
  - Changelogs contain the set of changes to the schema defined in YAML
- `db-changelog-master.xml` is where we store the list of all migrations that have been written as well as some common variables used throughout changelog files.
  - ***YOU CAN ONLY APPEND TO THE LIST.***
    - Imagine the changelog as the git history for our databases. If you pull out a commit or change one along the way. Everything after it will break/require changes.

Liquibase: Best Practices

If you need to do something in Liquibase but don't know the best way, you can get help/guidance from a variety of sources:

🧑‍🍳	Liquibase Recipes at Appian	A reviewed, curated list of best practices
🤓	Liquibase Reviewers room	The chat room where Appian's Liquibase experts are available
👨‍💻	Copy an existing changeset	Though beware not all historical changes have followed best practices
🪲 🚫	Troubleshooting Liquibase Migrations guide	A guide with some specific tips for troubleshooting debugging liquibase migrations.

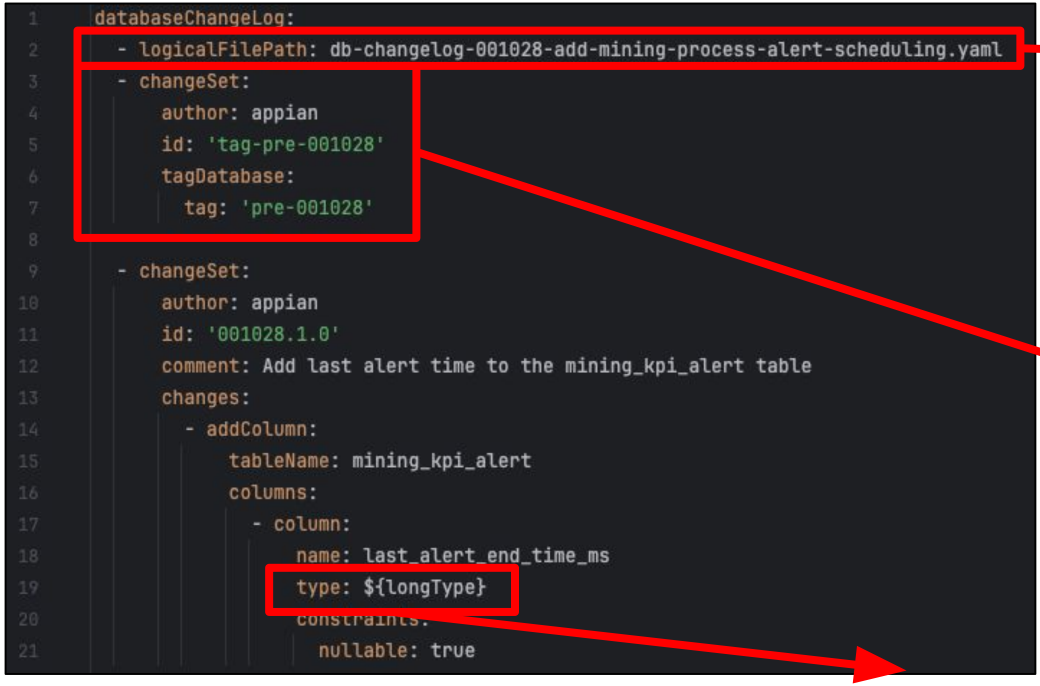
Liquibase: Deploying Changes

1. Make your new changelog file with a descriptive name.
2. Add the file to `db-changelog-master.xml`
3. Compile and Sync (if you have ear-resources synced, you're good!)
4. Restart appserver
   a. You must at least build so that changes to the file are inside of your docker container.
   b. The new columns will be added on appserver restart!

Liquibase: Migrating

- Migrations occur on AppServer startup
    - In the logs you will see: "`com.appiancorp.rdbms.hb.DataSourceManagerHbImpl - [jdbc/AppianAnywherePrimary] Checking schema and migrating if necessary (appian/db/changelog/db-changelog-master.xml)...`"
    - This step compares the changelog-master file's contents to what currently exists in the "DATABASECHANGELOG" table in the database.
        - This table is a source of truth for the actual changes that have been executed over time.
        - Each row corresponds to an individual `<changeSet>` tag within the YAML changelog files that will be detailed in the coming slides.
    - If anything was modified that already exists in the table, ERROR
    - Otherwise, apply the newly added changes in the non-existent changelog files.

Liquibase Changelog Eccentricities



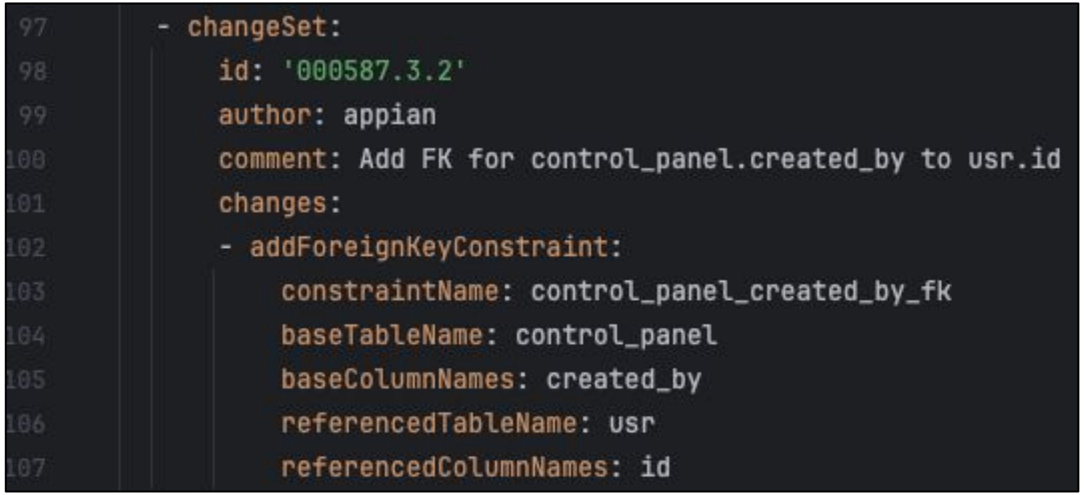
```yaml
databaseChangeLog:
  - logicalFilePath: db-changelog-001028-add-mining-process-alert-scheduling.yaml
  - changeSet:
      author: appian
      id: 'tag-pre-001028'
      tagDatabase:
        tag: 'pre-001028'

  - changeSet:
      author: appian
      id: '001028.1.0'
      comment: Add last alert time to the mining_kpi_alert table
      changes:
        - addColumn:
            tableName: mining_kpi_alert
            columns:
              - column:
                  name: last_alert_end_time_ms
                  type: ${longType}
                  constraints:
                    nullable: true
```

Yeah, unfortunately you have to tell liquibase that the file you're in is the file you're in.

The first change in a changelog is tagging the database table saying it is officially the end of the previous version …think `git tag`

As referred to earlier, this constant points to a value in `db-changelog-master.xml`
If you find yourself added an explicit type - **Don't!** There's probably one already defined.

Liquibase Changelogs: Foreign Keys



```
- changeSet:
    id: '000587.3.2'
    author: appian
    comment: Add FK for control_panel.created_by to usr.id
    changes:
    - addForeignKeyConstraint:
        constraintName: control_panel_created_by_fk
        baseTableName: control_panel
        baseColumnNames: created_by
        referencedTableName: usr
        referencedColumnNames: id
```

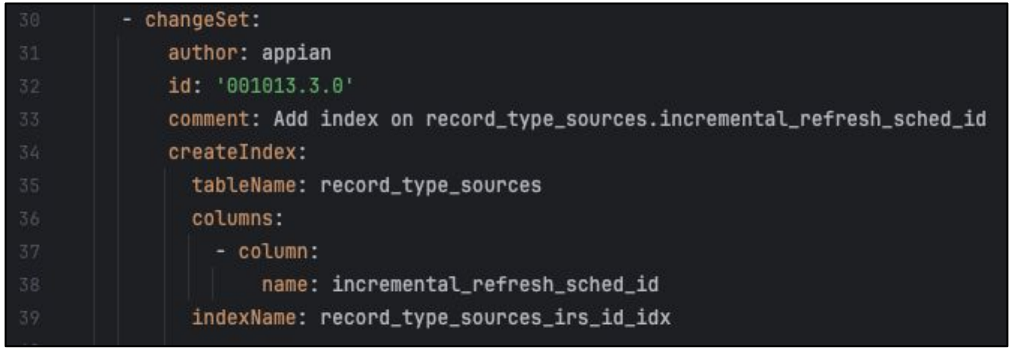
Foreign Keys are columns in a table whose value is sourced directly from a column in another table.

In the example here, we are creating this relationship between the user table and the created_by field on the control_panel.

Things to note:

- Constraint names should be of the format `<base-table-name>_<base-column-name-abbreviated-if-needed>_fk`
- Some database types limit the length of constraint names to 30 characters. Don't exceed.
- You must also create an index for each foreign key relationship (see next slide)

Liquibase Changelogs: Indices



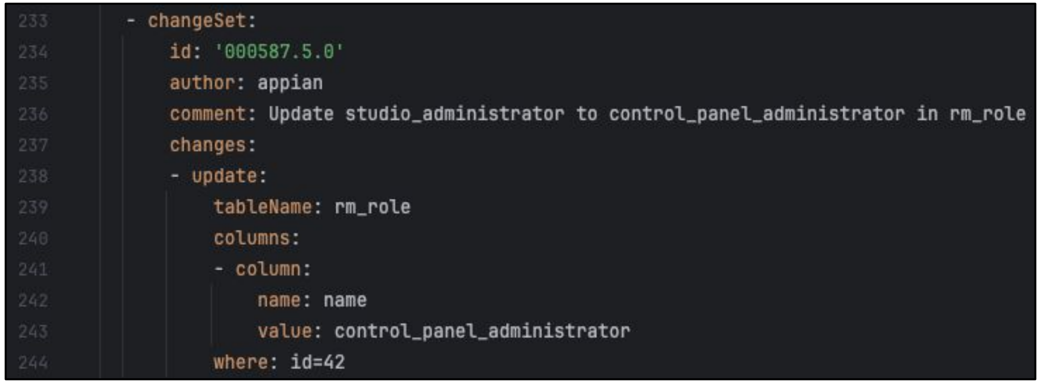
```yaml
    - changeSet:
        author: appian
        id: '001013.3.0'
        comment: Add index on record_type_sources.incremental_refresh_sched_id
        createIndex:
          tableName: record_type_sources
          columns:
            - column:
                name: incremental_refresh_sched_id
          indexName: record_type_sources_irs_id_idx
```

A database index improves performance of queries by reducing the amount of pages the DB has to look through to find a given row. We add indices for fields that will be commonly looked up (like foreign keys) to dramatically reduce search times. See this link for a more detailed explanation of indices (it's sql specific but the concepts are the same)

Things to note:

- Index names should be: `<table-name>_<column-name-abbreviated-if-needed>_idx`

Liquibase Changelogs: Updating Columns



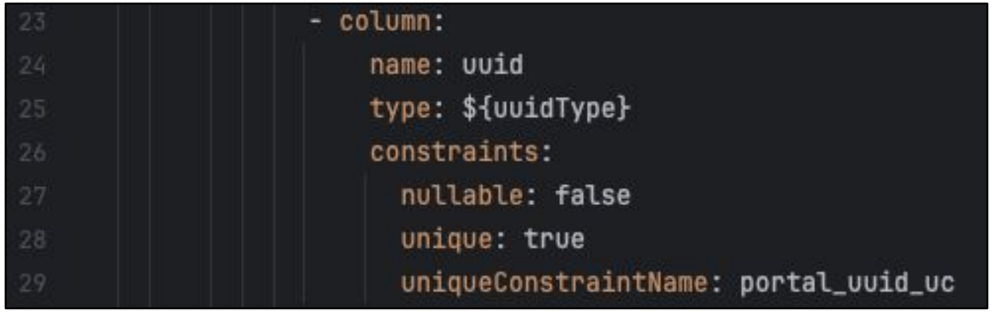
```yaml
- changeSet:
    id: '000587.5.0'
    author: appian
    comment: Update studio_administrator to control_panel_administrator in rm_role
    changes:
    - update:
        tableName: rm_role
        columns:
        - column:
            name: name
            value: control_panel_administrator
        where: id=42
```

Liquibase provides some changeset tags that allow you to update certain characteristics of a given table. This corresponds directly to the SQL UPDATE command. See this page for more details.

- The where statement needs to be valid SQL (quotes etc.)

Migrations that update data require that you write a migration test (stay tuned)

Liquibase Changelogs: Unique Constraints

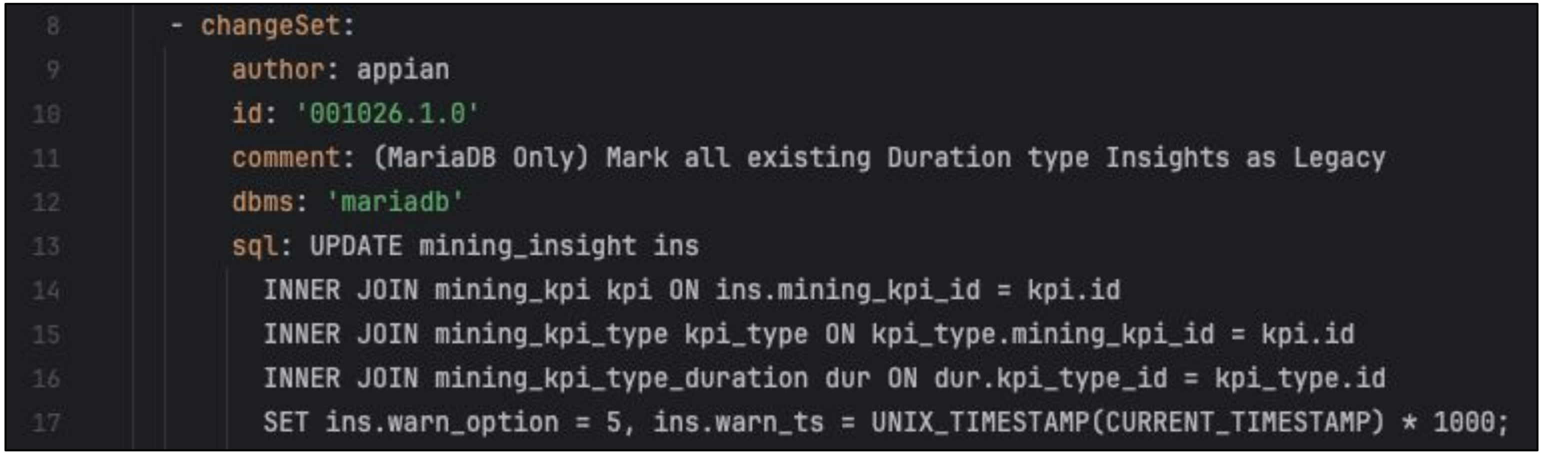


```
- column:
    name: uuid
    type: ${uuidType}
    constraints:
      nullable: false
      unique: true
      uniqueConstraintName: portal_uuid_uc
```

Unique Constraints require on the database level that all entries in the given column must be unique. We generally define them inline when we create the column. They can also be added as their own changeSet.

- Unique Constraints should be of the format:
  `<table-name>_<column-name-abbreviated-if-needed>_uc`
- They have the same length restriction (30 chars)

Liquibase Changelogs: Custom SQL



```
- changeSet:
    author: appian
    id: '001026.1.0'
    comment: (MariaDB Only) Mark all existing Duration type Insights as Legacy
    dbms: 'mariadb'
    sql: UPDATE mining_insight ins
      INNER JOIN mining_kpi kpi ON ins.mining_kpi_id = kpi.id
      INNER JOIN mining_kpi_type kpi_type ON kpi_type.mining_kpi_id = kpi.id
      INNER JOIN mining_kpi_type_duration dur ON dur.kpi_type_id = kpi_type.id
      SET ins.warn_option = 5, ins.warn_ts = UNIX_TIMESTAMP(CURRENT_TIMESTAMP) * 1000;
```

Sometimes we run into situations where we must modify/move data in a pre-existing table. You can do this by executing your own SQL statements as a changeSet. It can also be done in Java (outside the scope of this course).

- This must only include valid SQL, no database specific syntax or keywords. This is because the SQL must be compatible with all of the database types that Appian supports.

Liquibase Changelog Automated Tests

Database migration tests extend `DataSourceManagerSchemaMigrationTestBase` which provides setup and teardown as well as a few common utilities.

These tests must be written for any migration in which you have manipulated existing columns or data! This includes simple column additions. Put another way, a test should be added unless it's adding a new table that does not reference any existing tables.

If no test will be added, it requires explicit justification on the pull request.

Liquibase Changelog Automated Tests Cont.

Migration tests happen in 3 basic steps:

1. Setup pre-migration data
   - Calling `migrateToPrevious(String targetTag)` allows us to perform a migration to any previous migration tag. This should be the tag immediately preceding the one we're testing.
   - In order to populate the database, implement pre-migration versions of the RDBMS bean and DAO.
2. Perform the migration from the existing tag to the latest by calling `dataSourceManager.initialize()`.
3. Make assertions
   - Use the updated RDBMS bean and DAO to retrieve data from the migrated database
   - Make any necessary assertions against this data

How to change database on remote dev-Stub

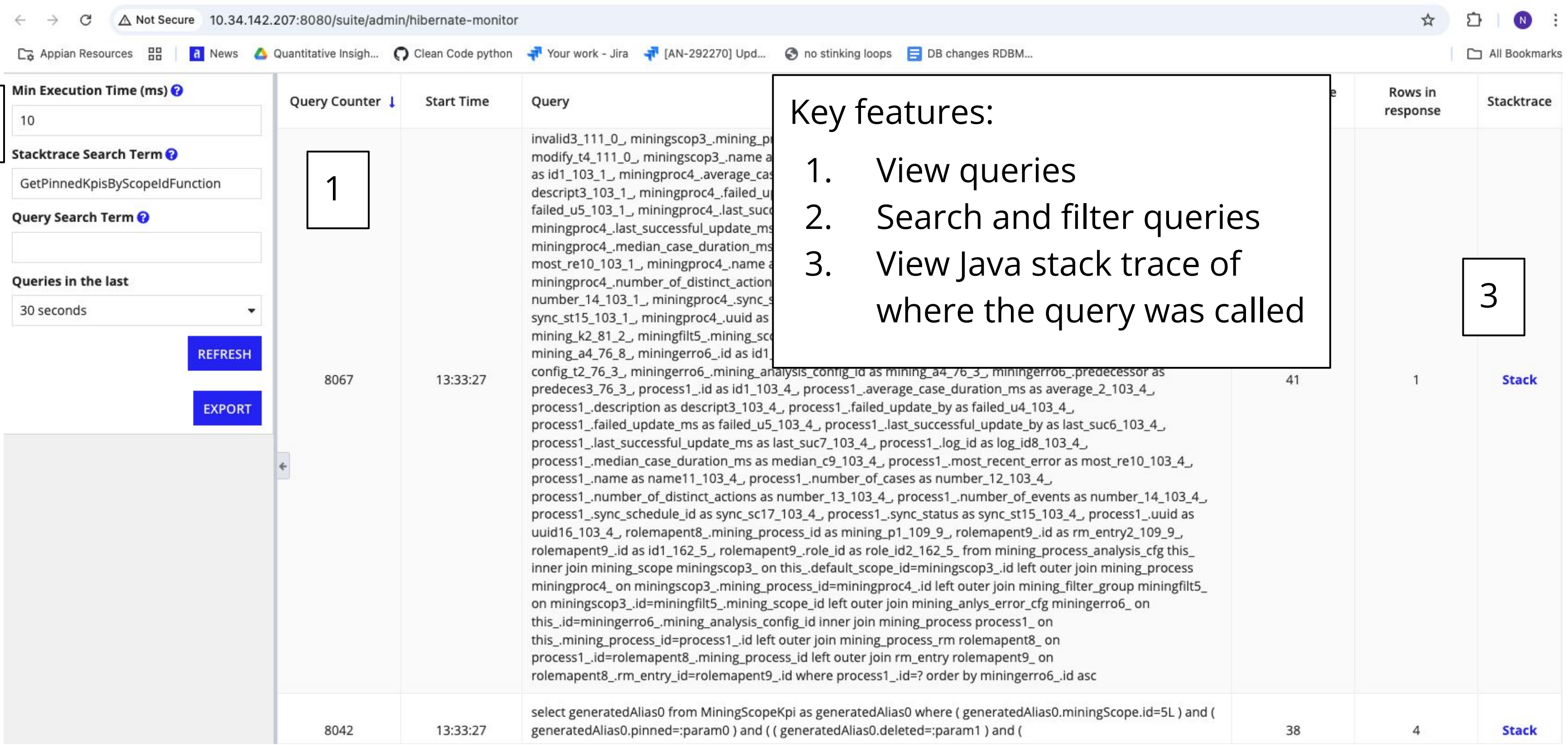
Stub, just so this command is searchable in drive

How to change database on remote dev

```
./dev config --rdbms {mariadb,mysql,sqlserver,oracle,postgresql,db2}
```

To run tests against another database type, follow the manual steps identified here

View hibernate query log on remote - Stub



⚠ Not Secure 10.34.142.207:8080/suite/admin/hibernate-monitor

Appian Resources | News | Quantitative Insigh... | Clean Code python | Your work - Jira | [AN-292270] Upd... | no stinking loops | DB changes RDBM... | All Bookmarks

2

Min Execution Time (ms)
10

Stacktrace Search Term
GetPinnedKpisByScopeIdFunction

Query Search Term

Queries in the last
30 seconds

REFRESH

EXPORT

1

3

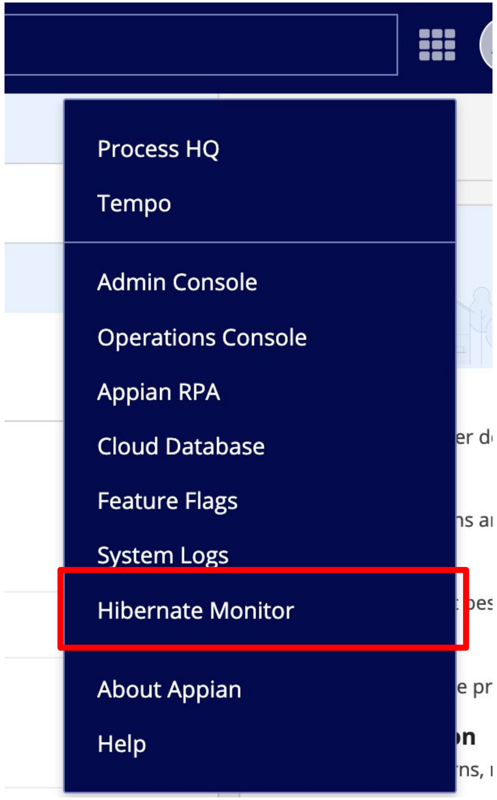
Query Counter ↓	Start Time	Query	e	Rows in response	Stacktrace
8067	13:33:27	invalid3_111_0_, miningscop3_.mining_pr modify_t4_111_0_, miningscop3_.name a as id1_103_1_, miningproc4_.average_cas descript3_103_1_, miningproc4_.failed_u failed_u5_103_1_, miningproc4_.last_succ miningproc4_.last_successful_update_ms miningproc4_.median_case_duration_ms most_re10_103_1_, miningproc4_.name a miningproc4_.number_of_distinct_action number_14_103_1_, miningproc4_.sync_s sync_st15_103_1_, miningproc4_.uuid as mining_k2_81_2_, miningfilt5_.mining_sc mining_a4_76_8_, miningerro6_.id as id1 config_t2_76_3_, miningerro6_.mining_analysis_config_id as mining_a4_76_3_, miningerro6_.predecessor as predeces3_76_3_, process1_.id as id1_103_4_, process1_.average_case_duration_ms as average_2_103_4_, process1_.description as descript3_103_4_, process1_.failed_update_by as failed_u4_103_4_, process1_.failed_update_ms as failed_u5_103_4_, process1_.last_successful_update_by as last_suc6_103_4_, process1_.last_successful_update_ms as last_suc7_103_4_, process1_.log_id as log_id8_103_4_, process1_.median_case_duration_ms as median_c9_103_4_, process1_.most_recent_error as most_re10_103_4_, process1_.name as name11_103_4_, process1_.number_of_cases as number_12_103_4_, process1_.number_of_distinct_actions as number_13_103_4_, process1_.number_of_events as number_14_103_4_, process1_.sync_schedule_id as sync_sc17_103_4_, process1_.sync_status as sync_st15_103_4_, process1_.uuid as uuid16_103_4_, rolemapent8_.mining_process_id as mining_p1_109_9_, rolemapent9_.id as rm_entry2_109_9_, rolemapent9_.id as id1_162_5_, rolemapent9_.role_id as role_id2_162_5_ from mining_process_analysis_cfg this_ inner join mining_scope miningscop3_ on this_.default_scope_id=miningscop3_.id left outer join mining_process miningproc4_ on miningscop3_.mining_process_id=miningproc4_.id left outer join mining_filter_group miningfilt5_ on miningscop3_.id=miningfilt5_.mining_scope_id left outer join mining_anlys_error_cfg miningerro6_ on this_.id=miningerro6_.mining_analysis_config_id inner join mining_process process1_ on this_.mining_process_id=process1_.id left outer join mining_process_rm rolemapent8_ on process1_.id=rolemapent8_.mining_process_id left outer join rm_entry rolemapent9_ on rolemapent8_.rm_entry_id=rolemapent9_.id where process1_.id=? order by miningerro6_.id asc	41	1	Stack
8042	13:33:27	select generatedAlias0 from MiningScopeKpi as generatedAlias0 where (generatedAlias0.miningScope.id=5L) and (generatedAlias0.pinned=:param0) and ((generatedAlias0.deleted=:param1) and (	38	4	Stack

Key features:

1. View queries
2. Search and filter queries
3. View Java stack trace of where the query was called

View hibernate query log on remote - Stub

- Can be accessed:
  - In `/design` from the waffle menu
  - By directly navigating to `/suite/admin/hibernate-monitor`
- Key features
  - View Queries
  - Search and filter queries
  - View Java stack trace of where the query was called
- Recommended use cases
  - Debugging
  - Performance
    - Can filter on long queries
    - Can easily identify overquerying
- Overview slides Indie time Spec



Additional Resources

Liquibase

Hibernate

RDBMS SOP

How to add a new designer object

How to persist new entity in primary data source

Finding Help

https://www.baeldung.com/ has good explanations of Hibernate basics

https://discourse.hibernate.org/c/hibernate-orm Discourse threads (this sometimes skews more recent versions though) - you can post questions.

The Appian Library also has a book that covers Hibernate in depth for any reference / understanding. It is for a previous version, but it is still helpful.

Congratulations!

You have now completed this course.