Introduction to Data Transformation with Code Repositories

Course Overview & Objectives

Purpose of the Course
Introduce engineers to Palantir Foundry's Code Repositories for building scalable data transformation workflows with production-grade code.

Learning Outcomes
Gain the ability to create repositories, manage branches, develop transforms, test and commit changes, and merge code into production.

Core Concepts
Understand foundational elements—Code Repositories, Data Lineage, Branching, Transforms, Pull Requests, and Master Branch workflows.

Practical Emphasis
Hands-on exercises and best practice applications to simulate real-world data engineering tasks.

Getting Started in Palantir Foundry

Understanding the Code Repository Workflow

Workflow Overview
Follow a nine-step cycle—Preview data, create repositories, branch safely, develop and test transformations, commit, pull request, and merge to production.

Prerequisites
Access to a Foundry instance, understanding of SQL/Python/Java, and appropriate repository permissions.

Accessing Code Repositories
Within a Foundry project, navigate to '+ New' → 'Code Repository', select a language template, and initialize with a descriptive name.

Development Setup
Foundry's web IDE enables collaborative coding, testing, and version control through Git-backed repositories.

Exploring Data Lineage in Foundry (Monocle)

Visualizing Data Dependencies and Flows

Understanding Data Lineage
Monocle displays datasets, transformations, and their relationships as an interactive graph—each node and edge representing data flow and dependencies.

Navigating the Interface
Users can expand upstream datasets, inspect schema and transformation code, and view data freshness and build status directly from the graph.

Benefits of Previewing Lineage
Enables early detection of missing data, outdated sources, and schema mismatches before starting development.

Integration with Repositories
Each dataset node links directly to its generating transformation in the Code Repository for quick debugging and traceability.

Creating a Code Repository

Initializing and Configuring Repositories in Foundry

Repository Initialization
Within your Foundry project, create a new Code Repository, select the preferred language template (SQL, Python, or Java), and name it descriptively.

Repository Structure
Automatically generated structure includes build configuration, documentation, and transform folders tailored to the selected language.

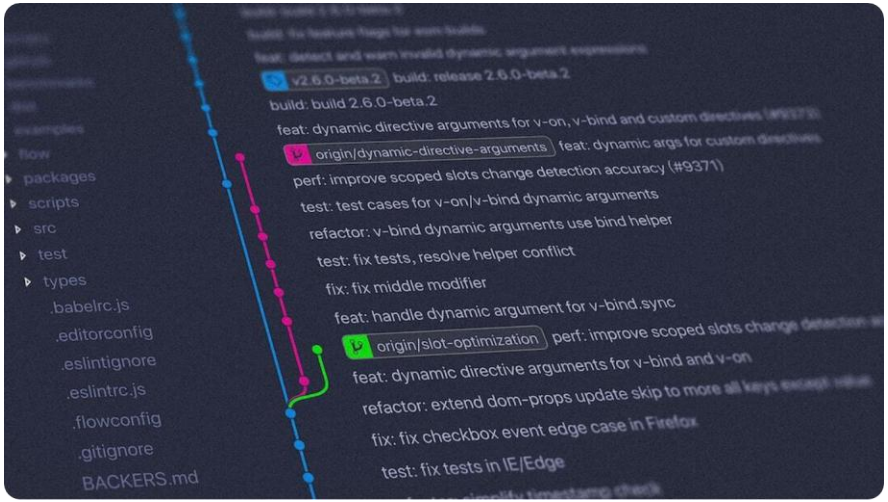
Branch Protection
The master branch is protected from direct edits, requiring all updates to go through Pull Requests—ensuring code quality and version integrity.

Configuration Best Practices
Use meaningful repository names, document purpose in README.md, and manage permissions for collaboration and security.

Branching Strategy

Safe and Collaborative Development Workflows

- **Purpose of Branching:** Branches enable parallel, isolated development without impacting the master branch, supporting collaboration and experimentation.
- **Creating Branches:** From the repository view, create a new branch named descriptively—e.g., feature/customer-cleaning or fix/data-quality.
- **Naming Conventions:** Adopt standard naming formats such as feature/, fix/, refactor/, or experiment/ to clarify intent and improve team coordination.
- **Branch Lifecycle:** Develop, test, and commit within your branch; merge back into master through pull requests once reviewed and validated.



Building an Identity Transform

Creating Your First Transformation in Foundry

Definition
An identity transform passes data from source to destination unchanged—serving as a baseline for pipeline validation and lineage setup.

SQL Implementation
Simple SQL example: SELECT * FROM `{project_path}/raw_customers` to replicate a raw dataset and test build functionality.

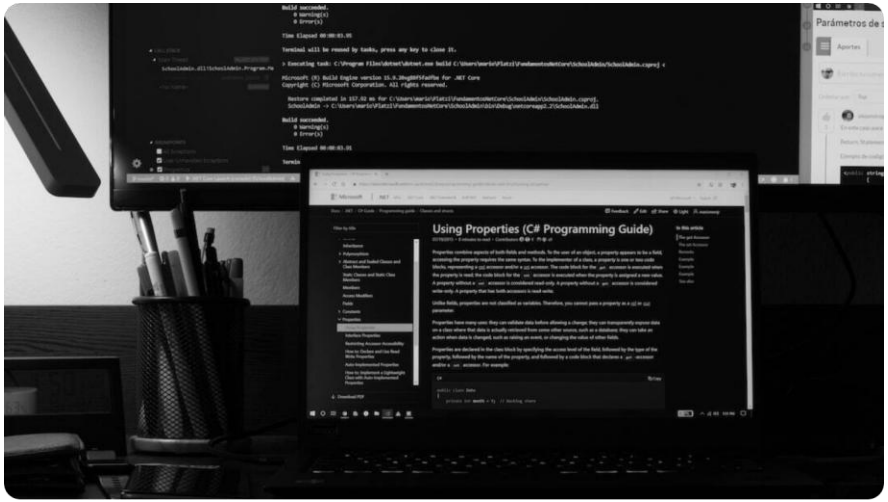
Python Implementation
Use @transform_df decorator in PySpark to define input/output datasets and return the raw DataFrame directly.

Purpose
Validates environment setup, confirms data accessibility, and ensures that transformations and builds are configured correctly.

Testing and Committing Code

Ensuring Transformation Accuracy and Version Control

- **Preview Mode:** Run code on sampled data to validate logic and syntax quickly without building full datasets, enabling faster iteration.
- **Committing Changes:** Each commit creates a labeled checkpoint, runs automated checks (syntax, schema, imports), and logs version history.
- **Commit Message Standards:** Use descriptive messages summarizing what and why—e.g., ‘Add identity transform for customer data’. Avoid vague terms like ‘fix’.
- **Handling Failures:** Review error logs in the Build helper, correct syntax or schema issues, and recommit once validated.



Building Your Dataset

From Preview to Production-Ready Outputs

- **Purpose of Building:** The build process materializes datasets from transformations, linking inputs and outputs in the Data Lineage graph for downstream use.
- **Preview vs. Build:** Preview tests logic on samples; Build executes the full transformation to generate complete datasets for production.
- **Monitoring Builds:** Track progress in the Build Helper—status icons indicate success (green), failure (red), or progress (blue).
- **Troubleshooting:** Resolve issues like timeouts or schema errors by optimizing transformations, filtering early, or adjusting compute profiles.

Merging and Pull Requests

Collaborative Code Review and Deployment

- **Purpose of Pull Requests:** PRs formalize code review, allowing contributors to propose, discuss, and validate changes before merging into master.
- **Creating a PR:** Ensure your branch is up to date, document changes clearly, and assign reviewers for approval before merge.
- **Review and Approval:** Reviewers assess code quality, functionality, and performance; approved PRs ensure production readiness.
- **Merge Strategies:** Foundry supports Merge Commit, Squash, and Rebase—Squash is preferred for cleaner history and traceability.

Preprocessing Data

Cleaning, Type Conversion, and Validation

- **Purpose of Preprocessing:** Transform raw data into clean, validated, and standardized datasets ready for analysis and downstream modeling.
- **Cleaning Utilities:** Reusable Python functions remove duplicates, trim whitespace, and handle null values consistently across datasets.
- **Type Utilities:** Standardize and validate data types using conversion functions (e.g., strings to integers, date parsing, boolean normalization).
- **Application in Transforms:** Integrate utilities within PySpark transforms for large-scale preprocessing, ensuring maintainability and modularity.

Best Practices and Troubleshooting

Ensuring Performance, Collaboration, and Data Quality

Development Discipline
Use descriptive names, commit frequently, and document code thoroughly to maintain clarity and traceability.

Collaboration Standards
Follow clear pull request guidelines—small, focused changes reviewed by peers to ensure consistency and maintainability.

Performance Optimization
Filter early, partition large datasets, and configure appropriate memory profiles for efficient builds.

Troubleshooting Common Issues
Address merge conflicts, build timeouts, and data quality failures by reviewing logs, validating inputs, and optimizing logic.