Securing Dataverse with an Adapted Command Design Pattern
An application design that enforces permission-based policies

- Supports declarative syntax for stating “Action X on object Y requires permissions $P_1..P_n$”
  - With dynamic overrides when needed
- Based on an adaptation of the classic Command Pattern
What we’ll cover

- Dataverse
  - Intro, Technology, Features, Background
- Adapting the Command Pattern
- The Pattern in Practice
Dataverse
Dataverse

- An open-source platform to publish, cite, and archive research data
- Built to support multiple types of data, users, and workflows
- Developed at Harvard’s Institute for Quantitative Social Science (IQSS) since 2006
- Development funded by IQSS and with grants, in collaboration with institutions around the world
- 15 on the core team - developers, designers, UI/UX, metadata specialists, curation manager
26 installations around the world
● 40+ code contributors outside of the Core Team
● Hundreds of members of the Dataverse Community - developers, researchers, librarians, data scientists
  ○ Dataverse Google Group
  ○ Dataverse Community Calls
  ○ Dataverse Community Meeting
Glassfish Server 4.1

Java SE8
Java EE7
- Presentation: JSF (PrimeFaces), RESTful API
- Business: EJB, Transactions, Asynchronous, Timers
- Storage: JPA (Entities), Bean Validation

Storage: Postgres, Solr, File System / Swift / S3

~95K Lines of code
Dataverse Features - Data

● Persistent IDs / URLs
  ○ DataCite / Handle
● Automatically Generated Citations with attribution
● Compliant with FAIR and data citation principles
● Domain-specific Metadata
● Versioning
● File Storage
  ○ Local / Swift (OpenStack) / S3 (Amazon)
Dataverse Features - Users

- Multiple Sign In options
  - Native
  - Shibboleth
  - OAuth (ORCID)
- Dataverses within Dataverses
- Branding
- Widgets
Dataverse Features - Workflows

- Permissions
- Access Controls and Terms of Use
- Publishing Workflows
- Private URLs
- Upload / Download Workflows
  - Browser
  - Dropbox
  - Rsync (for big data “packages”)
Dataverse Features - Interoperability

- APIs
  - SWORD
  - Native
- Harvesting (OAI-PMH)
  - Client
  - Server
What is a Dataverse or Dataset?
### Permissions / Roles

**Robust Permission System:**

- **System Roles**
- **Custom Roles (can be defined per installation)**
- **Groups**
  - Explicit
  - IP
  - Shibboleth
- **Inheritance**
  - dataverse -> dataset
  - dataset -> file

#### Admin
- A person who has all permissions for datavases, datasets, and files.
- Permissions: AddDataverse, AddDataset, ViewUnpublishedDataverse, ViewUnpublishedDataset, DownloadFile, EditDataverse, EditDataset, ManageDataversePermissions, ManageDatasetPermissions, PublishDataverse, PublishDataset, DeleteDataverse, DeleteDatasetDraft

#### Contributor
- For datasets, a person who can edit License + Terms, and then submit them for review.
- Permissions: ViewUnpublishedDataset, DownloadFile, EditDataset, DeleteDatasetDraft

#### Curator
- For datasets, a person who can edit License + Terms, edit Permissions, and publish datasets.
- Permissions: AddDataverse, AddDataset, ViewUnpublishedDataverse, ViewUnpublishedDataset, DownloadFile, EditDataset, ManageDataversePermissions, PublishDataset, DeleteDatasetDraft

#### Dataset Creator
- A person who can add datasets within a dataverse.
- Permissions: AddDataset

#### Dataverse + Dataset Creator
- A person who can add subdataverses and datasets within a dataverse.
- Permissions: AddDataverse, AddDataset

#### Dataverse Creator
- A person who can add subdataverses within a dataverse.
- Permissions: AddDataverse

#### File Downloader
- A person who can download a file.
- Permissions: DownloadFile

#### Member
- A person who can view both unpublished dataverses and datasets.
- Permissions: ViewUnpublishedDataverse, ViewUnpublishedDataset, DownloadFile
Adapting the Command Pattern
Adapting the Command Pattern

- **Commands** specify:
  - which permissions are required to execute them,
  - which model objects are involved
  - what is the context in which the command was submitted

- A **central command execution engine** that can verify that a command should indeed be executed prior to invoking it.

- Circumvents the need for sprinkling multiple permission checks at various places in the code
Adapting the Command Pattern

This design ensures that:

• Users are only able to perform actions they have permissions to perform

• Developers must explicitly state which permissions are required to run each command

• To the extent that static permissions declaration is used, the permissions required to run each command are declaratively documented in the code
“Encapsulate a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.”

GoF, 1994

- Thus enabling to use code as data
The **Command** Design Pattern

- **Business logic implemented in Command Subclasses**

- **Code Reuse**
  - Controllers become thinner, mostly preparing commands from view/network request and submitting them to the CommandEngine for execution

- **Security by Design**
  - Each command is annotated with requires permissions
  - Permissions tested by the engine - single place, single implementation
  - Auditing: Engine logs all executed commands

- **Testable**
public interface Command<R> {
    R execute(CommandContext ctxt) throws CommandException;
    Map<String, DvObject> getAffectedDvObjects();
    Map<String, Set<Permission>> getRequiredPermissions();
    DataverseRequest getRequest();
}

- `execute()` is where the work is done.
- Injected objects are made available via `CommandContext`
- `getAffectedDvObjects()` and `getRequiredPermissions()` detail which objects are affected and what permissions are needed to affect them
- `getRequest()` allows the permission system to detect which permissions the user has.
Sample Command: Delete a Role

```java
@RequiredPermissions(Permission.ManageDataversePermissions)
public class DeleteRoleCommand extends AbstractVoidCommand {

    private final DataverseRole doomed;

    public DeleteRoleCommand(DataverseRequest aRequest, DataverseRole doomed) {
        super(aRequest, doomed.getOwner());
        this.doomed = doomed;
    }

    @Override
    protected void executeImpl(CommandContext ctxt) throws CommandException {
        for (RoleAssignment ra : ctxt.roles().roleAssignments(doomed.getId())) {
            ctxt.roles().revoke(ra);
        }
        ctxt.roles().delete(doomed.getId());
    }
}
```
@DELETE
@Path("/{id}\")

public Response deleteRole( @PathParam("id") Long id ) {
    DataverseRole role = rolesSvc.find(id);
    if ( role == null ) {
        return notFound("role with id " + id + " not found");
    } else {
        try {
            execCommand(new DeleteRoleCommand(createDataverseRequest(findUserOrDie()), role));
            return okResponse("role " + id + " deleted.");
        } catch (WrappedResponse ex) {
            return ex.refineResponse("Cannot delete role " + id + ".");
        }
    }
}
Results of execCommand

Executing a command has 4 possible results:

- Issued command **makes no sense** (e.g. delete published data)
  - IllegalCommandException
  - Wrapped in WrappedResponse: 403 FORBIDDEN

- Issuing user is **not permitted** to perform the command
  - PermissionException
  - Wrapped in WrappedResponse: 401 UNAUTHORIZED

- General **Server Error**
  - CommandException
  - Wrapped in WrappedResponse: 500 INTERNAL_SERVER_ERROR
  - Plus, logging.

- **Works**
  - Return the result of the command (Java object)
Commands can be composed of other commands:

```java
@RequiredPermissions({})
public class GetLatestAccessibleDatasetVersionCommand extends AbstractCommand<DatasetVersion>{
    private final Dataset ds;

    @Override
    public DatasetVersion execute(CommandContext ctxt) throws CommandException {
        try {
            return ctxt.engine().submit(new GetDraftDatasetVersionCommand(getRequest(), ds));
        } catch (PermissionException ex) {
            return ctxt.engine().submit(new GetLatestPublishedDatasetVersionCommand(getRequest(), ds));
        }
    }
}
```
Dynamic Permissions

Permissions can be defined at runtime:

```java
@RequiredPermissions({})
public class GetDatasetCommand extends AbstractCommand<Dataset>{
    ...

    @Override
    public Map<String,Set<Permissions>> getRequiredPermissions(){
        if (dataset.isReleased()) {
            return Collections.emptySet();
        } else {
            return Collections.singleton(Permission.ViewUnpublishedDataset);
        }
    }
}
```
The Pattern in Practice

- Used in Dataverse since 2015
  - Largest effort was coding the engine and infrastructure
  - Provided to developers with a few basic examples
  - No discernible performance issues
- Discovered that we needed the ability to define the permissions at runtime
Benefits

• Straightforward to refactoring existing code
  o Basic commands are simple
  o Business logic already existed; just needed to be transferred

• Existing knowledge of annotations
  o Standard practice in Java

• Makes code maintenance easier
Challenges

- Motivating developers to use this pattern
  - In particular external developers
- Providing a good understanding of each of the different kinds of exceptions
- Ensuring proper use of command composition and dynamic permissions
  - In the beginning, wrote too many commands, instead of trying to find commonality
Securing Dataverse with an Adapted Command Design Pattern

Gustavo Durand
Michael Bar-Sinai
Merce Crosas