BARCELONA

DRUPALCON 2015

BARCELONA
DRUPALCON 2015

Entity storage, the Drupal 8 way

Francesco Placella

Coding and development - http://bit.ly/d8-esa

About me

- Francesco Placella, plach on drupal.org, from Venice, Italy
- Senior Performance Engineer at Tag1 Consulting
- Working with Drupal since 2006
- Maintainer of the core *Language system*
- Maintainer of the core *Language* and *Content Translation* modules
- *Unofficial* maintainer of the core Entity *storage*, Entity *form* and Entity *translation* subsystems
- http://twitter.com/plach__

Outline

- Drupal 7 vs Drupal 8
- Dealing with entity data
- Entity type and field definitions
- Storage schema
- Core SQL storage
- The fun stuff

Drupal 7 vs Drupal 8

Drupal 7

- Field *swappable storage*
- Field data can live in NoSQL storage, remote storage
- Every field is configured *independently*
- Possibly problematic for *entity querying*
- Supports only Fields, *properties* are always stored in the SQL database

Drupal 8

- Switched from field-based to *entity-based* storage
- Storage is still *swappable*
- Supports also *base fields* (e.g. the node type)
- *Entity querying* works nicely
- Fields can no longer be *shared* among entity types

  → you can have fields with the *same name* in different entity types

SQL is not dead, it just smells funny

Dealing with entity field data

- Swappable backends imply *storage-agnostic* code
- *Contrib* modules should not assume a SQL storage

  → either leverage the Entity CRUD API

  → or provide their own APIs (e.g. Views)
- *Custom* modules can assume a specific storage

  → should NOT bypass the Entity API

  → use SQL-specific APIs if needed

The Entity Query API

- To query entity field data we have the *Entity Query API*
  → the successor of the D7 Entity Field Query system
  → improved syntax → DBTNG
  → leverages swappable query backends
- Supports expressing *relationship* between entity types
  → the SQL backend translates those in `JOIN`s
- Supports expressing *aggregation* queries (!)
- Very powerful but obviously *not as expressive as SQL*

Legal SQL usages

- Always retrieve *identifiers*, also via custom SQL queries
  → do not retrieve partial data
- Always load an entity before accessing field data
- Always save an entity to write field data to the storage
- Bypassing the Entity API means you are on your own
  → unexpected behaviors, cache invalidation issues, …
- At least encapsulate SQL-specific code in a swappable service

It’s all a matter of definitions

Entity type definition

- An entity type *definition* (a plugin definition) describes the entity type to the system
- *Content* entities rely on field data
- *Configuration* entities use plain properties and are stored in configuration
- A definition has several properties allowing to customize the entity type’s behavior

Key definition properties

- The `handlers` section defines, among the rest:
  - the `storage` handler that performs CRUD entity operations
  - the `storage_schema` that manages the entity storage schema when needed (!)
- The `revisionable` and `translatable` properties may have an impact on how data is stored → schema

Entity Field API

- The D8 Entity Field API generalizes the D7 Field API
- Every piece of data attached to an entity is a *field*
- *Base fields* are shared among all available bundles (e.g. `nid`)
- *Bundle fields* may be attached only to certain bundles (e.g. `field_image`)
- Both are handled the same (e.g. Views or REST support)

Field definitions

- Base field definitions typically live in code

  → defined via `hook_entity_base_field_info()`

- Bundle field definitions typically live in configuration

  → defined via `hook_entity_bundle_field_info()`

  → the *Field* module allows to create bundle field definitions based on its configuration

  → can be defined in code too

Field storage definitions

- Field storage definitions collect the information required to *store* a field (surprise!)
- *Base* field definitions are usually instances of the `BaseFieldDefinition` class

  → both a field and a field storage definition
- *Bundle* field definitions share a field storage definition

  → can exist even when no bundle field has been defined

The Entity Storage Schema

Schema generation

- The storage handler is responsible for *managing its own schema*, if used
  → schema is *automatically generated* based on entity type and field definition
- Schema is created on module installation and dropped on uninstallation

Core SQL storage

- Generates tables for base and bundle fields
  - → single base fields are stored in *shared* tables
  - → bundle fields and multiple base fields are stored in *dedicated* tables
- Supports four different *shared table layouts* depending on
  - → entity type *translatability*
  - → entity type *revisionability*

Shared table layouts

- *Simple* entity types use

  → the base table to store base field data
- *Translatable* entity types use

  → the *base table* to store entity keys

  → the *data table* to store translated base field data
- *Revisionable* entity types use

  → the *base table* to store base field data

  → the *revision table* to store data for revisionable base fields

Shared table layouts

- Translatable and revisionable entity types use
  - → the *base table* to store entity keys
  - → the *data table* to store translated base field data
  - → the *revision table* to store entity key revisions and revision metadata
  - → the *revision data table* to stores translated base field revision data
- The storage schema supports switching between layouts

The Table Mapping API

- How to query shared tables?
  → via the *Entity Query API* (storage agnostic)
  → via the *Table Mapping API* (SQL-specific)
- The Table Mapping API allows to write SQL queries in a layout-agnostic fashion
  → It is used by Views to implement its SQL backend
  → Currently core entity type support only the DefaultTableMapping → assumes one of the previous layouts

Entity Updates

- Entity Updates leverage a dedicated API
- The Entity Definition Update Manager is able to detect any mismatch between the definitions and the actual schema

→ allows to apply individual updates

→ trigger events when an update is applied

→ refuses to proceed if the change requires a data migration

Entity Updates

- Typically Entity Schema updates are applied via *update functions*
- A Drush command is available (`drush entup`) to *apply any pending entity update*
  - → this should be used only during *development*
  - → should NOT be used to get rid of the *status report error* in production

The Right Way

Define …

- *Define* any field needed to implement the business logic
- Field data will be *loaded/stored* automatically
- Automatic module *integration* via the Entity Field API
  - → revisionability, translatability
  - → Views, REST, Rules, …
- Field definitions can opt out by marking themselves as having *custom storage* (not recommended)
  - → mainly used for computed fields

… and code!

- Core entity types provide interfaces making *business logic* explicit, e.g. `NodeInterface::isSticky()`

  → encapsulate the implementation

  → better integrated with IDEs

  → mark required data model

- It's a good practice to provide a *wrapper* for module-provided fields

Shut up and show me some code

A simple tracker

- Simple module (http://bit.ly/d8-esa-ex) to list:

  → users having created a published node

  → total amount of created nodes

  → title of most recently created node

- Direct querying has poor performance → denormalize

  → add two fields to the user entity type

  → update their values on C(R)UD events

A simple tracker

- Field definitions and installation
- The entity wrapper
- Service encapsulating business logic

  → on node creation → aggregate entity query

  → on node deletion → regular entity query

  → retrieve the user list → entity query relationship → display
- Performant and *fully portable*!

Seriously?

What's Left?

- Switching between shared table layouts is supported only by the API
  → https://www.drupal.org/node/2274017
- Define custom indexes for the entity storage schema
  → https://www.drupal.org/node/2258347
- When adding new fields an initial value may be needed
  → https://www.drupal.org/node/2346019
- Base field purging
  → https://www.drupal.org/node/2282119

Sprint: Friday

- Sprint with the Community on Friday
- We have tasks for every skillset
- Mentors are available for new contributors
- An optional Friday morning workshop for first-time sprinters will help you get set up
- Follow @drupalmentoring

As you may have guessed…

Conclusions

- Use the *Entity Field API* to define your data model and code your *business logic* on top of it

  → leverage fields to store data, avoid custom storage!

- Always *retrieve identifiers* and *load entities* to access field data

  → the *Entity Query API* is very powerful, use it whenever possible!

Useful links

- Entity Storage API blog post
  → https://drupalwatchdog.com/blog/2015/3/entity-storage-drupal-8-way
- Drupal 8 Entity API documentation
  → https://www.drupal.org/node/2143503
- The Table Mapping API reference
  → https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Entity!Sql!TableMappingInterface.php/interface/TableMappingInterface/8
  → https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Entity!Sql!DefaultTableMapping.php/class/DefaultTableMapping/8

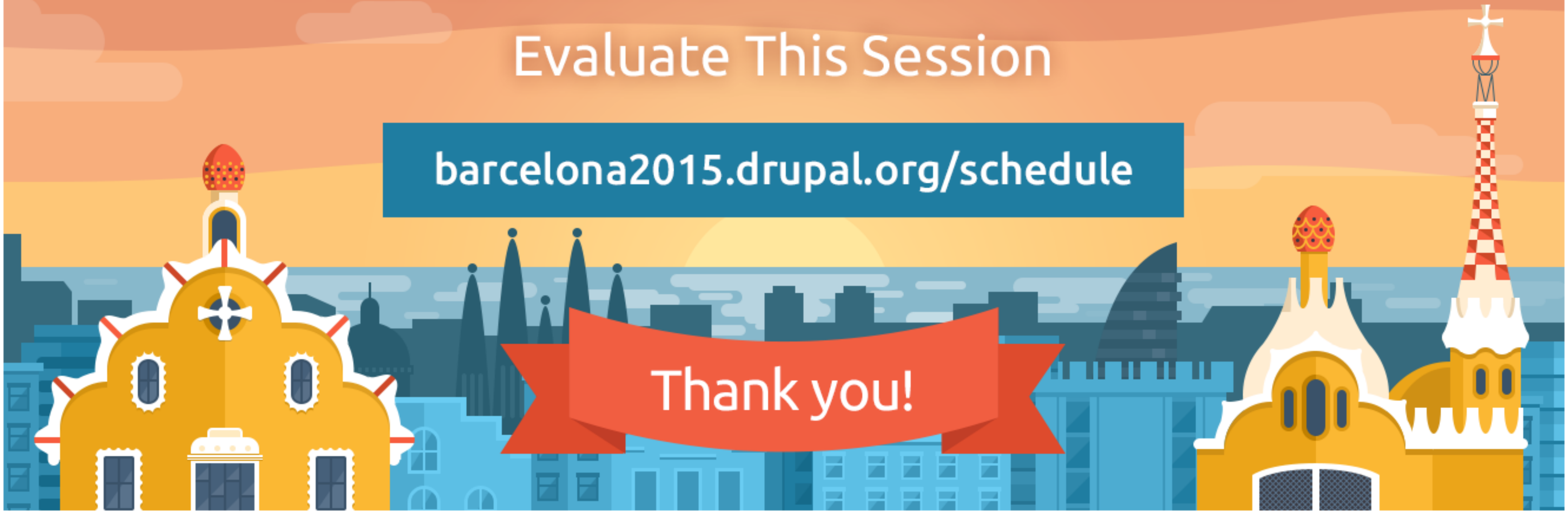
Question & Answers

BARCELONA
DRUPALCON 2015

What Did You Think?

Evaluate This Session

barcelona2015.drupal.org/schedule



Thank you!