{Perl School}

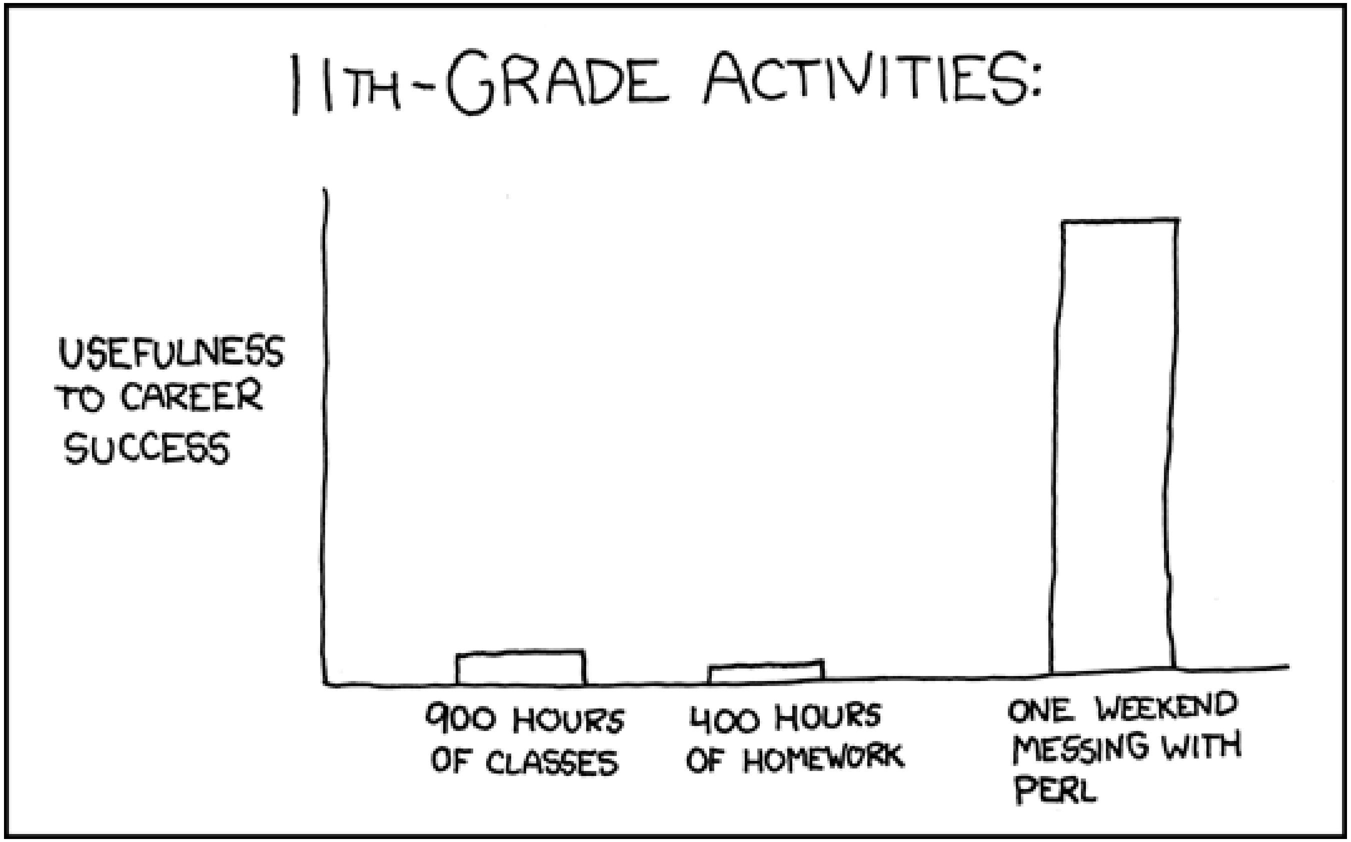
Database Programming with Perl and DBIx::Class

Dave Cross
dave@perlschool.co.uk

Perl School

- Low cost Perl training
- Training at all levels
- Trying to build a buzz about Perl
- Perl is not dead
- Perl is Modern

Xkcd Says


11TH-GRADE ACTIVITIES:
USEFULNESS TO CAREER SUCCESS
900 HOURS OF CLASSES
400 HOURS OF HOMEWORK
ONE WEEKEND MESSING WITH PERL

Your Help Please

- Trying to build a buzz about Perl
- You can help
- Please tell your friends
- Blog
- Twitter
- Facebook
- http://perlschool.co.uk

Upcoming Courses

- Perl School 5: Object Oriented Programming with Perl and Moose
  - 6th April 2013
- Perl School 6: Database Programming with Perl and DBIx::Class
  - 8th June 2013
- http://perlschool.co.uk/upcoming/

Admin Stuff

- Tickets
- Facilities
- Lunch
- Slides
- Feedback

Timings

- 10:00 Session 1
- 11:15 Break
- 11:30 Session 2
- 13:00 Lunch
- 14:00 Session 3
- 15:30 Break
- 15:45 Session 4
- 17:00 End

What We Will Cover

- Introduction to relational databases
- Introduction to databases and Perl
  - DBI
  - ORM
- Schema Classes
- Basic DB operations
  - CRUD

What We Will Cover

- Advanced queries
  - Ordering, joining, grouping
- Extending DBIC
- Deploying and updating schemas
- DBIC and Moose
- Further information

{Perl School}

Relational Databases

Relational Databases

- *A Relational Model of Data for Large Shared Data Banks*
  - Ted Codd (1970)
- Applying relational calculus to databases
- See also Chris Date
  - *Database in Depth* (2005)
  - *SQL and Relational Theory* (2011)
  - *Database Design and Relational Theory* (2012)

Relational Concepts

- Relation
  - Table
  - (Hence “relational”)
- Tuple
  - Row
- Attribute
  - Column

Some More Concepts

- Primary key
  - Unique identifier for a row within a table
- Foreign key
  - Primary key of a table that appears in another table
  - Used to define relationships between tables
  - e.g artist_id in a table containing CDs

Referential Integrity

- Check that database is in a meaningful state
  - No CDs without artist ID
  - No artist IDs that don't exist in the artist table
- Constraints that ensure you can't break referential integrity
  - Don't delete artists that have associated CDs
  - Don't insert a CD with a non-existent artist ID

SQL

- Structured Query Language
- Standard language for talking to databases
- Invented by IBM early 1970s
  - SEQUEL
- ISO/ANSI standard
- Many vendor extensions

DDL & DML

- Two sides of SQL
- Data Definition Language
  - Defines tables, etc
  - CREATE, DROP, etc
- Data Manipulation Language
  - Create, Read, Update, Delete data
  - CRUD
  - INSERT, SELECT, UPDATE, DELETE

{Perl School}

Databases and Perl

Talking to Databases

- Database vendors supply an API
- Usually a C library
- Defines functions that run SQL against a DB
- All vendors' APIs do the same thing
- All vendors' APIs are completely different

Ancient History

- Perl 4 had ways to link to external libraries
  - Like database APIs
- Static linking only
- Build a separate Perl binary for every database
  - oraperl, sybperl, etc
- Call API functions from Perl code

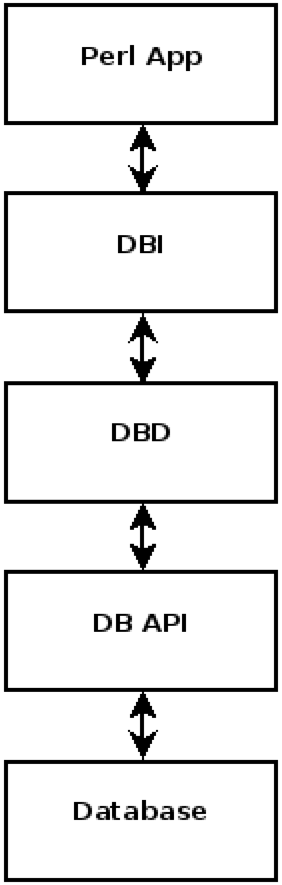
The Middle Ages

- Perl 5 introduced dynamic linking
- Load libraries at compile time
- Oraperl, Sybperl etc became CPAN modules
- `use Oraperl;`
- Still writing DB-specific code

Early Modern Era

- DBI.pm
- Standard database interface
- Database driver converts to API functions
  - DBD::Oracle, DBD::Sybase, etc
- Code becomes more portable
- (Except for vendor extensions)

DBI Architecture



DBI Architecture

- Programmer writes code to DBI spec
- DBD converts code to database API
- DBD converts Perl data structures as appropriate
- DBD converts returns data into Perl data structures

Loading DBI

- `use DBI;`
- No need to load specific DBD library
  - Sometimes DBD exports constants that you will need

Connecting to DB

- Communicate with database through a “database handle”

-
  ```
  my $dbh = DBI->connect(
    'dbi:mysql:host=foo.com:database=foo',
    $username, $password, \%options
  );
  ```
- Different DBDs have different options
- 'mysql' defines the DBD to load
  - DBD::mysql in this case

Selecting Data

- Select data using a prepare/execute/fetch cycle
-
  ```
  my $sql = 'select col1, col2 from some_tab';
  my $sth = $dbh->prepare($sql);
  $sth->execute;
  while (my $row = $sth->fetch) {
    say join ' : ', @$row;
  }
  ```

Inserting Data

- Insert data using a similar approach

-
  ```
  my $sql = 'insert into some_table (id, col1)
             values (1, “Foo”)';
  my $sth = $dbh->prepare($sql);
  $sth->execute; # No fetch required
  ```
- Or using do(...) shortcut

-
  ```
  $dbh->do($sql);
  ```

Updating and Deleting

- Update or delete data in exactly the same way

-
  ```
  my $sql = 'update some_table set col1 = “Bar”
             where id = 1';
  my $sth = $dbh->prepare($sql);
  $sth->execute;
  ```
- Or

-
  ```
  $dbh->do('delete from some_table
            where id = 1');
  ```

DBI Advantages

- Standard API for interacting with databases
- Programmer no longer needs to understand vendor APIs
  - Except the DBD author
- Increased programmer productivity
- Increased programmer flexibility

DBI Disadvantages

- Programmers still writing raw SQL
  - Which is boring
  - And error-prone
- DBI returns “dumb” data structures
  - Arrays or hashes
  - Often need to be converted into objects

DB Frameworks

- 10 years ago people started writing SQL generators
- Store a DB row in a hash
  - DBI has a fetchrow_hashref() method
- Generate SQL for simple CRUD operations

Next Steps

- Turn those hashes into objects
- Class knows table name
- Class knows column names
- Class knows primary key
- SQL generation moved into superclass
- All DB tables have an associated class

{Perl School}

Object Relational Mapping

Relational Database

- Consider database storage structures
- A table defines a type of data that can be stored
- A row is a single instance of that type of data
- A column is an attribute of that instance

Object Oriented

- Consider OO storage structures
- A class defines a type of data that can be stored
- An object is a single instance of that type of data
- An attribute is an attribute of that instance

ORM

- Database concepts and OO concepts map well onto each other
- A database table is a lot like an OO class
- A database row is a lot like an OO object
- A database column is a lot like an OO attribute
- We can use this to make our lives easier

ORM Principles

- A Object Relational Mapper converts between database data and objects
- In both directions
- Select data from the database and get an object back
- Change that object and update the database automatically

Replacing SQL

- Instead of

-
  ```
  SELECT *
  FROM   my_table
  WHERE  my_id = 10
  ```
- and then dealing with the prepare/execute/fetch code

Replacing SQL

- We can write
-
  ```
  use My::Object;

  # warning! not a real orm
  my $obj = My::Object->retrieve(10)
  ```
- Or something similar

Perl ORM Options

- Plenty of choices on CPAN
- Fey::ORM
- Rose::DB
- Class::DBI
- DBIx::Class
  - The current favourite

{Perl School}

DBIx::Class

DBIx::Class

- Standing on the shoulders of giants
- Learning from problems in Class::DBI
- More flexible
- More powerful

DBIx::Class Example

- Modeling a CD collection
- Three tables
- artist (id, name)
- cd (id, artist_id, title, year)
- track (id, cd_id, title, sequence)

{Perl School}

Defining Classes

DBIC Classes

- Two mandatory types of class
- One schema class
  - CD::Schema
- One result class for each table
  - CD::Schema::Result::Artist
  - CD::Schema::Result::CD
  - CD::Schema::Result::Track

Schema Class

- Define schema class
- CD/Schema.pm

-
  ```
  package CD::Schema;
  use strict;
  use warnings;
  use base qw/DBIx::Class::Schema/;

  __PACKAGE__->load_namespaces();

  1;
  ```

Result Classes

- Need one result class for every table
- Needs to know
  - The table name
  - The column names
  - The primary key
  - Relationships to other tables

Result Classes

- CD/Schema/Result/Artist.pm
-
  ```
  package CD::Schema::Result::Artist;
  use base qw/DBIx::Class::Core/;

  __PACKAGE__->table('artist');
  __PACKAGE__->add_columns( # simple option
    qw/ id name /
  );
  __PACKAGE__->set_primary_key('id');
  __PACKAGE__->has_many(
      'cds', 'CD::Schema::Result::CD',
      'artist_id'
  );
  1;
  ```

Result Classes

- CD/Schema/Result/CD.pm
-
  ```
  package CD::Schema::Result::CD;
  use base qw/DBIx::Class::Core/;

  __PACKAGE__->table('cd');
  __PACKAGE__->add_columns(
    qw/ id artist_id title year /
  );
  __PACKAGE__->set_primary_key('id');
  __PACKAGE__->belongs_to(
    'artist', 'CD::Schema::Result::Artist',
    'artist_id'
  );
  __PACKAGE__->has_many(
    'tracks', 'CD::Schema::Result::Track', 'cd_id'
  );
  1;
  ```

Result Classes

- CD/Schema/Result/Track.pm

-
  ```
  package CD::Schema::Result::Track;
  use base qw/DBIx::Class::Core/;

  __PACKAGE__->table('track');
  __PACKAGE__->add_columns(
    qw/ id cd_id title sequence /
  );
  __PACKAGE__->set_primary_key('id');
  __PACKAGE__->belongs_to(
    'cd', 'CD::Schema::Result::CD', 'cd_id'
  );

  1;
  ```

Defining Columns

- At a minimum you must define column names
- But you can give more information

-
  ```
  __PACKAGE__->add_columns(
    id => {
      data_type => 'integer',
      is_auto_increment => 1,
    },
    name => {
      data_type => 'varchar',
      size => 255,
    }
  );
  ```

Defining Relationships

- We have seen has_many and belongs_to
- Both ends of a many-to-one relationship
- Most common type of relationship
- Artists to CDs
- CDs to tracks
- Manager to employees
- Invoice to invoice lines
- Simple foreign key relationship

Other Relationships

- has_one
  - Only one child record
  - Person has one home address
- might_have
  - Optional has_one relationship
- Affects the SQL that is generated

Don't Repeat Yourself

- *The Pragmatic Programmer* says “Don't repeat yourself”
- Only one source for every piece of information
- We are breaking this rule
- We have repeated data

Repeated Information

-
  ```
  CREATE TABLE artist (
    artistid INTEGER PRIMARY KEY,
    name     TEXT NOT NULL
  );
  ```

Repeated Information

-
  ```
  package CD::Schema::Result::Artist;
  use base qw/DBIx::Class::Core/;

  __PACKAGE__->table('artist');
  __PACKAGE__->add_columns( # simple option
    qw/ id name /
  );
  __PACKAGE__->set_primary_key('id');
  __PACKAGE__->has_many(
      'cds', 'CD::Schema::Result::CD',
      'artist_id'
  );
  1;
  ```

Don't Repeat Yourself

- Information is repeated
- Columns and relationships defined in the database schema
- Columns and relationships defined in class definitions

Don't Repeat Yourself

- Need to define one canonical representation for data definitions
- Generate the other one
- Let's choose the DDL
- Generate the classes from the DDL

Database Metadata

- Some people don't put enough metadata in their databases
- Just tables and columns
- No relationships. No constraints
- You may as well make each column VARCHAR(255)

Database Metadata

- Describe your data in your database
- It's what your database is for
- It's what your database does best

DBIC::Schema::Loader

- DBIx::Class::Schema::Loader
  - Separate distribution on CPAN
- Creates classes by querying your database metadata
- No more repeated data
- We are now DRY
- Schema definitions in one place

dbicdump

- DBIC::Schema::Loader comes with a command line program called dbicdump

-
  ```
  $ dbicdump CD::Schema dbi:mysql:database=cd root ''
  Dumping manual schema for CD::Schema to
  directory . ...
  Schema dump completed.
  ```

-
  ```
  $ find CD
  CD
  CD/Schema
  CD/Schema/Result
  CD/Schema/Result/Cd.pm
  CD/Schema/Result/Artist.pm
  CD/Schema/Result/Track.pm
  CD/Schema.pm
  ```

{Perl School}

Simple CRUD

Loading DBIC Libraries

- Load the main schema class
- `use CD::Schema;`
- The load_namespaces call takes care of loading the rest of the classes

Connecting to DB

- The DBIC equivalent of a database handle is called a schema
- Get one by calling the connect method

-
  ```
  my $sch = CD::Schema->connect(
    'dbi:mysql:database=cd', $user, $pass
  );
  ```
- Connection parameters passed through to DBI

Inserting Data

- Interact with tables using a resultset object
- The schema class has a resultset method that will give you a resultset object
- `my $art_rs = $sch->resultset('Artist');`

Inserting Artists

- Use the create method on a resultset to insert data into a table
-
  ```
  my @artists = ('Elbow',
                 'Arcade Fire');

  foreach (@artists) {
    $art_rs->create({ name => $_ });
  }
  ```
- Pass a hash reference containing data
- Handles auto-increment columns

Inserting Artists

- The create method returns a new artist object
  - Actually a CD::Schema::Result::Artist

-
  ```
  my $bowie = $art_rs->create({
    name => 'David Bowie'
  });
  ```
- Result objects have methods for each column
- `say $bowie->id;`

Inserting Artists

- An alternative is to use the populate() method
-
  ```
  my @artists = $art_rs->populate(
    [ 'name' ],
    [ 'Arcade Fire' ],
    [ 'Elbow' ],
  );
  ```
- Pass one array reference for each row
- First argument is a list of column names

Insert Related Records

- Easy to insert objects related to existing objects

-
  ```
  $bowie->add_to_cds({
    title => 'The Next Day',
    year => 2013
  });
  ```
- Foreign key added automatically
- add_to_cds method added because of relationships

Reading Data

- Selecting data is also done through a resultset object
- We use the search() method
-
  ```
  my ($bowie) = $art_rs->search({
    name => 'David Bowie'
  });
  ```

Reading Data

- There's also a find() method
- Use when you know there's only one matching row
- For example, using primary key

-
  ```
  my $bowie = $art_rs->find({
    id => 3,
  });
  ```

-
  ```
  my $bowie = $art_rs->find(3);
  ```

Searching Relationships

- Defining relationships allows us to move from object to object easily
-
  ```
  my $cd_rs = $sch->resultset('CD');
  my ($cd) = $cd_rs->search({
    title => 'The Seldom Seen Kid'
  });
  say $cd->artist->name; # Elbow
  ```
- The artist() method returns the associated artist object

Searching Relationships

- This works the other way too

-
  ```
  my ($artist) = $art_rs->search({
                   name => 'Elbow',
                 });

  foreach ($artist->cds) {
    say $_->title;
  }
  ```
- The cds() method returns the associated CD objects

What Search Returns

- The search() method returns different things in different contexts
- In list context it returns a list of result objects that it has found
- In scalar context it returns another resultset
  - That only contains the matching result objects

What Search Returns

-
  ```
  my $artist = $art_rs->search({
    name => 'Elbow';
  });
  ```
  - $artist is a resultset object

-
  ```
  my ($artist) = $art_rs->search({
    name => 'Elbow';
  });
  ```
  - $artist is a result object

Taming Search

- To get all of the result objects from a resultset call its all() method

-
  ```
  my $artist = $art_rs->search({
    name => 'Elbow';
  })->all;
  ```
  - $artist is a **result** object

Taming Search

- To get always get a resultset, use search_rs() instead of search()

-
  ```
  my ($artist) = $art_rs->search_rs({
    name => 'Elbow';
  });
  ```
  - $artist is a **resultset** object

Updating Data

- Once you have a result object you can change any of its attributes
- `$bowie->name('Thin White Duke');`
- Use the update() method to save it to the database
- `$bowie->update();`

Updating Data

- You can also call update() on a resultset

-
  ```
  my $davids = $art_rs->search({
    name => { like => 'David %' },
  });

  $davids->update({
    name => 'Dave',
  });
  ```

Deleting Data

- Deleting works a lot like updating
- Delete a single record

-
  ```
  my ($britney) = $art_rs->search({
    name => 'Britney Spears'
  });

  $britney->delete;
  ```

Deleting Data

- You can also delete a resultset
-
```
my $cliffs = $art_rs->search({
  name => { like => 'Cliff %' }
});

$cliffs->delete;
```

Cascading Deletes

- What if any of the artists have CDs in the database?
- They get deleted too
- Referential integrity
- Prevent this by changing relationship definition

-
  ```
  __PACKAGE__->has_many(
    'cds', 'CD::Schema::Result::CD', 'artistid',
    { cascade_delete => 0 },
  );
  ```

Insert Multiple Records

- Create can be used to insert many rows
-
  ```
  $art_rs->create({
    name => 'Arcade Fire',
    cds => [{
      title => 'The Suburbs'
    },
    {
      title => 'Funeral'
    }]
  });
  ```

Find or Insert

- Insert an object or return an existing one

-
  ```
  my $killers = $art_rs->find_or_create({
    name => 'The Killers'
  });
  ```
- **Note:** Need a unique index on one of the search columns

Update or Create

- Update an existing object or create a new one

-
  ```
  my $killers = $art_rs->update_or_create({
    name => 'The Killers'
  });
  ```
- **Note:** Need a unique index on one of the search columns

Transactions

- Transactions protect the referential integrity of your data
- Chunk of work that must all happen
- Temporary workspace for DB changes
- Commit or rollback at the end

Transactions & DBIC

- Schema object has a txn_do() method
- Takes a code reference as a parameter
- Adds BEGIN and COMMIT (or ROLLBACK) around code
- Transactions can include Perl code

Transactions & DBIC

-
  ```
  $schema->txn_do( sub {
    my $obj = $rs->create(\%some_obj);
    $obj->add_to_children(\%some_child);
  });
  ```

Advanced Searches

Advanced Searches

- search() can be used for more complex searchs
- See SQL::Abstract documentation for full details

AND

- Use a hash reference to combine conditions using AND

-
  ```
  $person_rs->search({
    forename => 'Dave',
    email => 'dave@perlschool.co.uk'
  });
  ```

-
  ```
  WHERE forename = 'Dave'
  AND email = 'dave@perlschool.co.uk'
  ```

OR

- Use an array reference to combine conditions using OR

-
  ```
  $person_rs->search([{
    forename => 'Dave'
  }, {
    email => 'dave@perlschool.co.uk'
  }]);
  ```

-
  ```
  WHERE forename = 'Dave'
  OR email = 'dave@perlschool.co.uk'
  ```

Combinations

- Combine hash references and array references for more flexibility
-
  ```
  $person_rs->search([{
    forename => 'Dave',
    username => 'dave'
  }, {
    email = 'dave@perlschool.co.uk'
  }]);
  ```

Many Values for Column

- Use an array reference to test many values for a column

-
  ```
  $person_rs->search({
    forename => [ 'Dave', 'David' ]
  });
  ```

-
  ```
  WHERE forename = 'Dave'
  OR forename = 'David'
  ```

Using SQL

- SQL::Abstract supports some SQL options

-
  ```
  $person_rs->search({
    forename => { like => 'Dav%' }
  });
  ```
- `WHERE forename LIKE 'Dav%'`

Using SQL

- More SQL-like options

-
  ```
  $person_rs->search({
    forename => {
      '-in' => [ 'Dave', 'David' ]
    }
  });
  ```
- `WHERE forename IN ('Dave', 'David')`

Using SQL

- More SQL-like options

-
  ```
  $person_rs->search({
    birth_year => {
      '-between' => [ 1970, 1980 ]
    }
  });
  ```

-
  ```
  WHERE birth_year
  BETWEEN 1970 AND 1980
  ```

Extra Search Attributes

- All of our examples have used one parameter to search
- `$rs->search(\%where_clause)`
- Search takes an optional second parameter
- Defines search attributes
- `$rs->search(\%where_clause, \%attrs)`

Select Specific Columns

- Default search selects all columns in a table
  - Actually all attributes in the class
- Use the columns attribute to change this

-
  ```
  $person_rs->search({
    forename => 'Dave'
  }, {
    columns => [ 'me.forename',
                 'me.surname' ]
  });
  ```
- Note table aliases

Add Columns

- You can invent columns and add them to the returned object
-
  ```
  $person_rs->search({
    forename => 'Dave'
  }, {
    +columns => {
        namelen => { length => 'me.forename' }
    }
  });
  ```
- Use get_column() to access this data
-
  ```
  $person->get_column('namelen')
  ```

Ordering Data

- Use search attributes to order the data
-
```
$person_rs->search({
  forename => 'Dave'
}, {
  order => { '-asc' =>
               [ 'me.surname' ] }
});
```

Paging

- Select a subset of the data
-
```
$person_rs->search({
    forename => 'Dave',
}, {
    rows => 10,
    page => 2
});
```
- You probably want to sort that query

Joining Tables

- Use the join attribute to join to other tables

-
  ```
  $art_rs->search({}, {
    columns => [ 'me.name', 'cds.title' ],
    join => [ 'cds' ]
  });
  ```
- Join artist table to CD table
- Return artist name and CD title

Aggregate Functions

- Use SQL aggregate functions like COUNT, SUM and AVERAGE

-
  ```
  $person_rs->search({}, {
    columns => [ 'me.forename',
                 name_count => {
                   count => 'me.forename'
                 } ],
    group_by => [ 'me.forename' ]
  });
  ```
- Use get_columns() to get the count

Join and Aggregate

- Combine joins and aggregates
-
  ```
  $art_rs->search({}, {
    columns => [ 'me.name',
                 cd_count => {
                   count => 'cds.id'
                 } ],
    group_by => [ 'me.forename' ],
    join => [ 'cds' ]
  });
  ```

Chaining Resultsets

- We said that search() can return a resultset
- We can call search() again on that resultset to further specify the search
- And so on...

Chaining Resultsets

-
  ```
  my $daves = $person_rs->search({
    forename => 'Dave'
  });

  my $women => $daves_rs->search({
    sex => 'F'
  });

  foreach ($women->all) {
    say $_->forename, ' ', $_->surname;
  }
  ```

Executing Resultsets

- A resultset is the definition of a query
- The query isn't run until you execute the resultset
- By calling all(), first(), next(), etc
  - `$person_rs->all`
- By calling search() in list context

  -
    ```
    @daves = $person_rs->search({
      forename => 'Dave',
    });
    ```

{Perl School}

More on Result Classes

Result Classes

- Result classes are usually generated by DBIx::Class::Schema::Loader
- Define columns
- Define relationships
- But we can add our own code to these classes

Derived Columns

- Sometimes it's handy to have a “column” that is derived from other columns
- Just add a method
-
```
sub name {
  my $self = shift;

  return $self->forename, ' ',
         $self->surname;
}
```

Actions

- Add methods defining actions that your class needs to carry out

-
  ```
  sub marry {
    my $self = shift;
    my $spouse = shift;

    $self->spouse($spouse->id);
    $spouse->spouse($self->id);
  }
  ```

Column Inflation

- Inflate a column into a more useful class when reading from database
- Deflate object into string before saving to database
- e.g. Convert datetime column to DateTime object

DateTime Inflation

- This is a standard feature of DBIC
- DBIx::Class::InflateColumn::DateTime
- Load as a component

  -
    ```
    __PACKAGE__->load_component(
      'DBIx::Class::InflateColumn::DateTime'
    );
    ```
- Define column as datetime

  -
    ```
    __PACKAGE__->add_columns(
      birth => { datatype => 'datetime' }
    );
    ```

DateTime Inflation

-
  ```
  my $person = $person_rs->first;

  my $birth = $person->birth;

  say ref $birth; # DateTime

  say $birth->day_name;
  ```

-
  ```
  $person_rs->create({
    name => 'Some Person',
    birth => DateTime->now
  });
  ```

DBIC::Schema::Loader

- Use the -o command line option to include components in generated classes
- `dbicdump -o components='[“InflateColumn::DateTime”]' ...`
- Adds the load_components() call to the classes

Manual Inflation

- You can define your own inflation/deflation code
- Use the inflate_column() method
-
  ```
  __PACKAGE__->inflate_column(
    'column_name' => {
      inflate_column => sub { ... },
      deflate_column => sub { ... },
    }
  );
  ```

Unicode Inflation

- Databases store strings as a series of bytes
- Well-behaved Unicode-aware code converts bytes to characters as the string enters the program
  - And vice versa
- Many DBDs have a flag to do this automatically
- Some don't

Unicode Inflation

-
  ```
  use Encode;
  __PACKAGE__->inflate_column(
    'some_text_column' => {
      inflate_column => sub {
        return decode('utf8', $_[0]);
      },
      deflate_column => sub {
        return encode('utf8', $_[0]);
      },
    }
  );
  ```

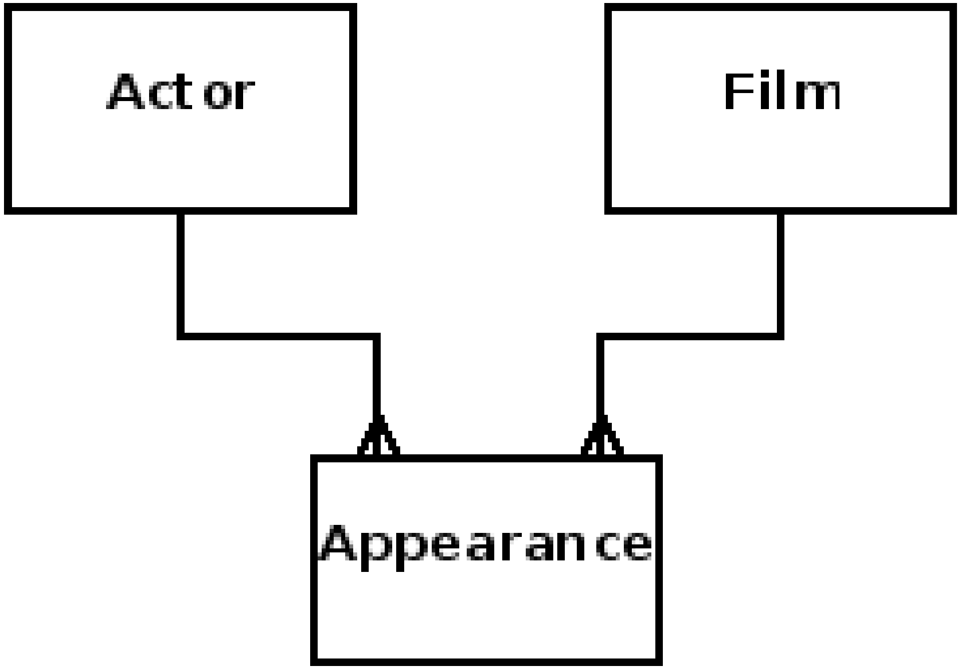
Relationships

- DBIx::Class::Schema::Loader generates many kinds of relationships from metadata
- It doesn't recognise many-to-many relationships
  - Linking tables
- We can add them manually in the result class

Many to Many

- An actor appears in many films
- A film features many actors
- How do you model that relationship?
- Add a linking table
  - Appearance
- Two foreign keys

Many to Many



Many to Many

- DBIx::Class::Schema::Loader finds the standard relationships
  - Actor has many Appearances
  - Appearances belong to Actor
  - Film has many Appearances
  - Appearances belong to Film
- We can add a many to many relationship
  - In both directions

Many to Many

-
  ```
  Film::Schema::Result::Actor->many_to_many(
    'films',  # new relationship name
    'appearances', # linking relationship
    'film'    # FK relationship in link table
  );

  Film::Schema::Result::Film->many_to_many(
    'actors', # new relationship name
    'appearances', # linking relationship
    'actor',  # FK relationship in link table
  );
  ```

Without Many to Many

-
  ```
  my $depp = $actor_rs->search({
    name => 'Johnny Depp'
  });

  foreach ($depp->appearances) {
    say $_->film->title;
  }
  ```

With Many to Many

-
  ```
  my $depp = $actor_rs->search({
    name => 'Johnny Depp'
  });

  foreach ($depp->films) {
    say $_->title;
  }
  ```

Editing Result Classes

- Editing result classes is useful
- But result classes are usually generated
  - DBIx::Class::Schema::Loader
- How do we regenerate classes?
- Without overwriting our additions

MD5 Hash

- A generated result class contains an MD5 hash

-
  ```
  # Created by DBIx::Class::Schema::Loader
  v0.05003 @ 2010-04-04 13:53:54
  # DO NOT MODIFY THIS OR ANYTHING ABOVE!
  md5sum:IvAzC9/WLrHifAi0APmuRw
  ```
- Add anything below this line
- Code below this line is preserved on regeneration

Resultset Classes

- We've looked a lot at editing result classes
- You can also edit resultset classes
- Often to add new search methods
- But resultset classes don't exist as files
- Need to create them first

Resultset Class

-
  ```
  package App::Schema::Resultset::Person

  use strict;
  use warnings;

  use base 'DBIx::Class::Resultset';

  1;
  ```

Default Search Values

-
  ```
  sub search_men {
    my $self = shift;

    return $self->search({
      sex => 'M'
    });
  }
  ```

Default Search Values

-
  ```
  sub search_men {
    my $self = shift;
    my ($cols, $opts) = @_;

    $cols ||= {};
    $opts ||= {};
    $cols->{sex} = 'M';
    return $self->search(
      $cols, $opts
    );
  }
  ```

Default Search Options

-
  ```
  sub search_sorted {
    my $self = shift;

    return $self->search({}, {
      order_by => 'name ASC'
    });
  }
  ```
- Similar changes for full version

{Perl School}

Extending DBIC

Extending DBIC

- DBIC is powerful and flexible
- Most of the time it can be made to do what you want
- Sometimes you need to change its default behaviour
- Override default methods

Overriding Methods

- Overriding methods is a standard OO technique
- Method in a subclass replaces one in a superclass
- Define subclass method with same name
- Subclass method has new behaviour

Overriding Methods

- Often the subclass behaviour needs to happen in addition to the superclass behaviour
- Subclass method needs to call the superclass method
- Ugly syntax
- `$self->SUPER::method()`

Overriding Methods

-
  ```
  sub do_something {
    my $self = shift;

    ...

    $self->SUPER::do_something(@_);

    ...
  }
  ```

Class::C3 / mro

- DBIC uses a non-standard method resolution technique
- mro
  - Method resolution order
- Specifically its Class::C3 implementation
- “better consistency in multiple inheritance situations”

Class::C3 / mro

- All you really need to know
- When overloading DBIC methods, use $self->next::method instead of SUPER
-
  ```
  sub do_something {
    my $self = shift;
    ...
    $self->next::method(@_);
    ...
  }
  ```

Overriding new()

- Result classes don't include a new method
- That's defined in the DBIx::Class superclass
- We can override it
-
```
sub new {
  my $class = shift;

  # do stuff

  return $self->next::method(@_);
}
```

Overriding new()

- Defaults for missing attributes
-
  ```
  sub new {
    my $class = shift;
    my $obj = shift;

    # Set birthday if it's missing
    $obj->{birth} ||= DateTime->now;

    # Superclass method does real work
    return $self->next::method($obj);
  }
  ```

Overriding update()

- Add audit information

-
  ```
  sub update {
    my $self = shift;

    # Set audit columns
    $self->upd_time(DateTime->now);
    $self->upd_by($Curr_User);

    # Superclass method does real work
    $self->next::method();
    say $self->name, ' updated';
  }
  ```

Overriding delete()

- Don't really delete rows

-
  ```
  sub delete {
    my $self = shift;

    # Set deleted flag
    $self->deleted(1);

    # Don't call superclass method!
    $self->update;
  }
  ```

DBIC and Moose

- Moose is the future of OO Perl
- Moose makes OO Perl easier, more powerful and more flexible
- Moose supports use alongside non-Moose classes
  - MooseX::NonMoose
- We can use DBIC with Moose

Write Your Own Classes

-
  ```
  package CD::Schema::Result::Artist;

  use Moose;
  use MooseX::NonMoose;
  extends 'DBIx::Class:Core';

  __PACKAGE__->table('artist');
  __PACKAGE__->add_columns(...);
  __PACKAGE__->set_primary_key(...);

  # define relationships
  ...

  __PACKAGE__->meta->make_immutable;
  ```

Write Your Own Classes

-
  ```
  package CD::Schema::Result::Artist;

  use Moose;
  use MooseX::NonMoose;
  extends 'DBIx::Class:Core';

  __PACKAGE__->table('artist');
  __PACKAGE__->add_columns(...);
  __PACKAGE__->set_primary_key(...);

  # define relationships
  ...

  __PACKAGE__->meta->make_immutable;
  ```

Using Moose Class

- As far as the user (i.e. the application programmer) is concerned there is no difference
- The same code will work
- `my $artist_rs = $schema->resultset('Artist');`
- `my $artist = $art_rs->create(\%artist);`
- `$artist->update;`
- `$artist_rs->search();`

Using Moose Class

- For the programmer writing the class, life gets better
- We now have all of the power of Moose
- Particularly for overriding methods
- Method modifiers

Method Modifiers

- More flexible and powerful syntax for overriding methods
- More control over interaction between subclass method and superclass method
- Easier syntax
  - No $self->SUPER::something()
  - No $self->next::method()

Overriding new()

- Run subclass method before superclass method

-
  ```
  before new  => sub {
    my $class = shift;
    my $obj = shift;

    # Set birthday if it's missing
    $obj->{birth} ||= DateTime->now;

    # Superclass method run
    # automatically
  }
  ```

Overriding update()

- Run subclass method around superclass method

-
  ```
  around update => sub {
    my $orig = shift;
    my $self = shift;

    # Set audit columns
    $self->upd_time(DateTime->now);
    $self->upd_by($Curr_User);

    # Superclass method does real work
    $self->$orig(@_);
    say $self->name, ' updated';
  }
  ```

Overriding delete()

- Run subclass method in place of superclass method

-
  ```
  override delete => sub {
    my $self = shift;

    # Set deleted flag
    $self->deleted(1);

    # Don't call superclass method!
    $self->update;
  }
  ```

Adding Roles

- Moose roles are pre-packaged features that can be added into your class
- Like mixins or interfaces in other OO languages
- Added with the keyword “with”

Role Example

-
  ```
  package App::Schema::Result::SomeTable;

  use Moose;
  use MooseX::NonMoose;

  extends 'DBIx::Class::Core';
  with 'Some::Clever::Role';
  ```

DBIC::Schema::Loader

- DBIx::Class::Schema::Loader has built-in support for Moose
- use_moose option
- With dbicdump

-
  ```
  $ dbicdump -o use_moose=1 CD::Schema \
    dbi:mysql:database=cd root ''
  ```
- Creates classes with the Moose lines included

{Perl School}

Deploying Schemas

Changing Schemas

- Database schemas change over time
- Tables added
- Columns added
- Column definitions change
- DBIC has tools to manage that

Don't Repeat Yourself

- We have two definitions of our database schema
- DDL
  - CREATE TABLE, etc
- DBIC
  - Perl code
- Choose one as canonical source

DDL vs DBIC

- We can create DBIC code from DDL
  - DBIx::Class::Schema::Loader
- We can create DDL from DBIC
  - $schema->deploy()

Deploy

- Schema objects have a deploy() method
- Generates DDL
  - Using SQL::Translator
  - Applies it to connected database
- Can also see the DDL
  - deployment_statements()
  - create_ddl_dir()

Schema Versions

- Versions change over time
- Need to cope with that
- Add a version to our schema class
- Set $VERSION

Schema Versions

-
  ```
  package CD::Schema;
  use warnings;
  use strict;
  use base 'DBIx::Class::Schema';

  our $VERSION = '0.01';

  __PACKAGE__->load_namespaces();

  1;
  ```

Schema Versions

-
  ```
  package CD::Schema;
  use warnings;
  use strict;
  use base 'DBIx::Class::Schema';

  our $VERSION = '0.01';

  __PACKAGE__->load_namespaces();

  1;
  ```

create_ddl_dir

- The create_ddl_dir() method is clever
- Given a previous version of a schema
- It can create ALTER TABLE statements
-
  ```
  $schema->create_ddl_dir(
    [ 'MySQL' ], $curr_ver,
    $directory, $preversion
  );
  ```
- This will be very useful

Deploying Versions

- DBIC includes a module called DBIx::Class::Schema::Versioned
- Upgrades schemas

DBIC::Sch::Versioned

- More changes to your schema class

-
  ```
  package MyApp::Schema;
  use base qw/DBIx::Class::Schema/;

  our $VERSION = 0.001;

  __PACKAGE__->load_namespaces;

  __PACKAGE__->load_components(
    qw/Schema::Versioned/
  );

  __PACKAGE__->upgrade_directory(
  '/path/to/upgrades/'
  );
  ```

DBIC::Sch::Versioned

- More changes to your schema class

-
  ```
  package MyApp::Schema;
  use base qw/DBIx::Class::Schema/;

  our $VERSION = 0.001;

  __PACKAGE__->load_namespaces;

  __PACKAGE__->load_components(
    qw/Schema::Versioned/
  );

  __PACKAGE__->upgrade_directory(
  '/path/to/upgrades/'
  );
  ```

Create Upgrade DDL

-
  ```perl
  use Getopt::Long;
  use CD::Schema;

  my $preversion, $help;
  GetOptions(
    'p|preversion:s'  => \$preversion,
  ) or die;

  my $schema = MyApp::Schema->connect(...);

  # continued...
  ```

Create Upgrade DDL

- `my $sql_dir = './sql';`
- `my $version = $schema->schema_version();`

-
  ```
  $schema->create_ddl_dir(
    'MySQL', $version, $sql_dir,
    $preversion
  );
  ```
- Creates all the DDL you need
  - Includes versioning tables

Upgrade DB

-
  ```
  use CD::Schema;
  my $schema = CD::Schema->connect(...);

  if ($schema->get_db_version()) {
    # Runs all the upgrade SQL
    $schema->upgrade();
  } else {
    # Schema is unversioned
    # Installs empty tables
    $schema->deploy();
  }
  ```

Better Tool

- DBIC::Schema::Versioned is part of the standard DBIC package
- DBIC::DeploymentHandler is a separate CPAN package
- More powerful
- More flexible

DBIC::DeploymentHndlr

- Advantages
  - Upgrades and downgrades
  - Multiple SQL files in one upgrade
  - Use Perl scripts for upgrade
- Disadvantages
  - Dependency hell

{Perl School}

Replication

Replication

- Some databases allow multiple copies of the same data
- Server software keeps replicants in step
- This can aid performance
- Different clients can talk to different servers
- Data on some replicants can lag

Types of Replication

- Master-Slave
  - One writeable copy of the database
  - Many readable replicants
  - e.g. MySQL

Types of Replication

- Multiple Master
  - Many writeable copies
  - Potential for deadlocks
  - e.g. Sybase

DBIC & Replication

- DBIC has beta support for master/slave replication
- Directs all writes to master connection
- Directs all reads to slave connection

DBIC & Replication

- Set the storage_type attribute on our schema object

-
  ```
  my $schema = CD::Schema->connect(...);

  $schema->storage_type([
    '::DBI::Replicated',
    { balancer => 'Random' },
  ]);
  ```

Add Slaves

- Add slave connections
-
  ```
  $schema->storage->connect_replicants(
    [$dsn1, $user, $pass, \%opts],
    [$dsn2, $user, $pass, \%opts],
    [$dsn3, $user, $pass, \%opts],
  );
  ```

Use Schema

- Use schema as usual
- Reads are delegated to a random slave
- Writes are delegated to the master
- You can force a read to the master

-
  ```
  $rs->search({ ... },
    { force_pool => 'master' });
  ```
  - Avoid race conditions

{Perl School}

Further Information

Documentation

- Lots of good DBIC documentation
  - perldoc DBIx::Class
  - perldoc DBIx::Class::Manual
- DBIx::Class::Manual::SQLHackers
  - Separate documentation distribution

Support

- Web site
  - http://www.dbix-class.org/
- Mailing list
  - See support page on web site
- IRC channel
  - #dbix-class on irc.perl.org

Books

- Good coverage in *The Definitive Guide to Catalyst*
  - Not completely up to date
- DBIC book being written
  - Schedule unknown

That's All Folks

- Any Questions?

{Perl School}

- Any Questions?