
DUBLIN
DRUPALCON

How do I Typed Data?

Matthew Radcliffe

Coding and Development

https://events.drupal.org/dublin2016/sessions/how-do-i-typed-data-drupal-8

Coding and Development

https://events.drupal.org/dublin2016/sessions/how-do-i-typed-data-drupal-8

Matthew Radcliffe

Kosada

Drupal professional, enthusiast since 2007.

mradcliffe

@mattkineme

I like hacking on things.
I like integrating with APIs.
I like figuring out the unknown.
I have opinions.

Summary

1. Concept of Typed Data and use cases.

2. Risks and benefits of using Typed Data directly rather than Entity API.

3. Working with Data Definitions and Data Types.

4. Working with Typed Data through services.

5. Unit tests with Typed Data.

Typed Data

- Typed Data API is a meta for primitive and composite data types in Drupal that are not otherwise defined, and provides a strongly-typed interface that can be passed into various sub-systems.

Typed Data

- Typed Data API is a meta for primitive and composite data types in Drupal that are not otherwise defined, and provides a strongly-typed interface that can be passed into various sub-systems.
  - Primitive data type: a basic or built-in type of a programming language.
  - Composite data type: any data type made up of primitive and other composite data types i.e. structs, classes.

Typed Data

- Typed Data API is a meta for primitive and composite data types in Drupal that are not otherwise defined, and provides a strongly-typed interface that can be passed into various sub-systems.
  - Primitive data type: a basic or built-in type of a programming language.
  - Composite data type: any data type made up of primitive and other composite data types i.e. structs, classes, associative arrays, annotations.
- In other words, Typed Data is the low level API to describe data within Drupal.

Use Cases for Typed Data Data Types

- Describe data defined elsewhere i.e. schemas from external systems.
  - Education, Media, Accounting, CRM, etc…
- Define special snowflake data types.

Example: Xero Integration

- Xero accounting platform with a Restful API.
  - Post invoices from Open Atrium cases. Invoice done. Click send. Back to coding.
  - Post bank transactions from Commerce/Ubercart payment transactions. Auto-matching Bank Statements to Bank Transactions is huge cost savings for a small company.
  - Do not necessarily need to store data twice. It's already stored off-site.

Escaping the Drupal Procedural Reality

- Make OAuth signed requests via drupal_http_request or straight up Curl (xero_query).
- Drupal form validation for bulk operations (xero_form_helper).
- Model data as associative arrays or strongly-coupled entities (xero_make, entity metadata).
- And sometimes I have to maintain external PHP libraries I do not want to maintain…

```
        $filterid = ( count($arguments) > 0 ) ? strip_tags(strval($arguments[0])) : false;
        if(isset($arguments[1])) $modified_after = ( count($arguments) > 1 ) ? str_replace( 'X','T', date( 'Y-m-dXH:i:s', strtotime($arguments[1])) ) :
false;
        if(isset($arguments[2])) $where = ( count($arguments) > 2 ) ? $arguments[2] : false;
        if ( is_array($where) && (count($where) > 0) ) {
           $temp_where = '';
           foreach ( $where as $wf => $wv ) {
              if ( is_bool($wv) ) {
                 $wv = ( $wv ) ? "%3d%3dtrue" : "%3d%3dfalse";
              } else if ( is_array($wv) ) {
                 if ( is_bool($wv[1]) ) {
                    $wv = ($wv[1]) ? rawurlencode($wv[0]) . "true" : rawurlencode($wv[0]) . "false" ;
                 } else {
                    $wv = rawurlencode($wv[0]) . "%22{$wv[1]}%22" ;
                 }
              } else {
                 $wv = "%3d%3d%22$wv%22";
              }
              $temp_where .= "%26%26$wf$wv";
           }
           $where = strip_tags(substr($temp_where, 6));
        } else {
           $where = strip_tags(strval($where));
        }
        $order = ( count($arguments) > 3 ) ? strip_tags(strval($arguments[3])) : false;
        $acceptHeader = ( !empty( $arguments[4] ) ) ? $arguments[4] : '';
        $method = $methods_map[$name];
        $xero_url = self::ENDPOINT . $method;
        if ( $filterid ) {
           $xero_url .= "/$filterid";
        }
        if ( isset($where) ) {
           $xero_url .= "?where=$where";
        }
        if ( $order ) {
           $xero_url .= "&order=$order";
        }
        $req  = OAuthRequest::from_consumer_and_token( $this->consumer, $this->token, 'GET',$xero_url);
        $req->sign_request($this->signature_method , $this->consumer, $this->token);
```

So how do I get there?

- Goal: Make HTTP Requests to integrate with external system
- Want:
  - Describe Xero types into something Drupal understands.
    - Primitive and complex (composite) data types.
    - Flexible enough to handle custom use case of not relying on any storage concerns.

Drupal 8: Age of Discovery

- Guzzle, OAuth 1.0 subscriber plugin, amongst other libraries available to pull in via Composer.

10:45 Thur: https://events.drupal.org/dublin2016/sessions/composer-based-workflows-drupal-8

Drupal 8: Age of Discovery

- Guzzle, OAuth 1.0 subscribe plugin, amongst other libraries available to pull in via Composer.
- Serialization module via Symfony Serializer component fit into the Guzzle ecosystem.

Drupal 8: Age of Discovery

- Guzzle, OAuth 1.0 subscribe plugin, amongst other libraries available to pull in via Composer.
- Serialization module via Symfony Serializer component fit into the Guzzle ecosystem.
- Typed Data is the API that serialization module uses to normalize data.

Xero Data Types

- Accounts
- Invoices
- Payments
- Credit Notes
- Contacts
- Users
- Bank Transactions
- Amount
- Line Items
- and more…

Xero Data Types

- Accounts
- Invoices
- Payments
- Credit Notes
- Contacts
- Users
- Bank Transactions
- Amount
- Line Items
- and more…

Xero Data Types

- Accounts
- Invoices
- Payments
- Credit Notes
- Contacts
- Users
- Bank Transactions
- Amount
- Line Items
- and more…

Xero Data Types

- Accounts
- Invoices
- Payments
- Credit Notes
- Contacts
- Users
- Bank Transactions
- Amount
- Line Items
- and more…

Is Typed Data a Good Fit?

- Why Typed Data?
  - Flexible means of describing data types defined outside of Drupal.
  - Typed Data API gets all of the things, and is injectable into Forms, Route Controllers, Normalizers, etc…
  - So that can work well with Core and Contrib modules such as Serialization (and most likely Rules).

What about Entity API?

- Why not Typed Data?
  - Typed Data Manager only provides instantiation of data types and definitions.
    - Entity Managers (Repositories) provide based on Entity annotation:
    - No form and access handler out of the box.
    - No caching/cache tags out of the box.
  - Data Definition for a composite data type is required and seems out of place.

Is Typed Data a Good Fit?

- I think that Entity API has a lot of baggage.
  - But some of it is overridable or ignorable (storage, form handlers).
  - But will other contrib modules cooperate?
- Implement data types as low-level data representations for API modules and leave the storage and other concerns to higher-level systems and modules.

How to Read a Map

- There's one complex data type that does not describe an entity
  - Map: the new associative array.
    - Implements \IteratorAggregate.

- Next challenge: how to create the elusive data type and data definition classes.

Data Type Plugin

- Data Type is a plugin class. Extends TypedData.
  - Plugin? What's a plugin? All the things.
- Data Type
  - Constructor usually inherited. Sets property values according to definition.
  - Add useful methods to your class.

mradcliffe, dixon_. 2015. Data type plugins. https://www.drupal.org/node/1795908.

```php
<?php

namespace Drupal\typed_example\Plugin\DataType;

use Drupal\Core\TypedData\Plugin\DataType\Map;

/**
 * @DataType(
 *   id = "typed_example_color",
 *   label = @Translation("Example Color"),
 *   definition_class = "\Drupal\typed_example\TypedData\ColorDefinition",
 *   list_class = "\Drupal\typed_example\Plugin\DataType\ExampleItemList"
 * )
 */
class Color extends Map {}
```

annotation type

plugin id

data definition

list data type

Data Definitions

- A complex data types requires definition classes to describe its properties.
  - Definitions offer
    - Constraints e.g. Regex, Choice
    - Label
    - Data typing

mradcliffe. 2015. Data Definitions. https://www.drupal.org/node/2487754.

```php
class ColorDefinition extends ComplexDataDefinitionBase {

  public function getPropertyDefinitions() {
    if (!isset($this->propertyDefinitions)) {
      $info = &$this->propertyDefinitions;

      $info['red'] = DataDefinition::create('float')
        ->setLabel('Red')
        ->setRequired(TRUE)
        ->addConstraint('Range', ['min' => 0, 'max' => 255]);

      $info['green'] = DataDefinition::create('float')
        ->setLabel('Green')
        ->setRequired(TRUE)
        ->addConstraint('Range', ['min' => 0, 'max' => 255]);

      $info['blue'] = DataDefinition::create('float')
        ->setLabel('Blue')
        ->setRequired(TRUE)
        ->addConstraint('Range', ['min' => 0, 'max' => 255]);
    }
    return $this->propertyDefinitions;
  }
}
```

```php
class ExampleDefinition extends ComplexDataDefinitionBase {

  public function getPropertyDefinitions() {
    if (!isset($this->propertyDefinitions)) {
      $info = &$this->propertyDefinitions;
      $info['name'] = DataDefinition::create('string')
        ->setLabel('Name')
        ->setDescription('A string property that is required.')
        ->setRequired(TRUE);

      $info['primary'] = ColorDefinition::create('typed_example_color')
        ->setLabel('Primary Color')
        ->setDescription('A complex data type as a child property.');

      $info['secondary'] = ListDataDefinition::create('typed_example_color')
        ->setLabel('Secondary Colors')
        ->setDescription('A list of complex data types as a child property.');
    }

    return $this->propertyDefinitions;
  }

}
```

List Data Definition

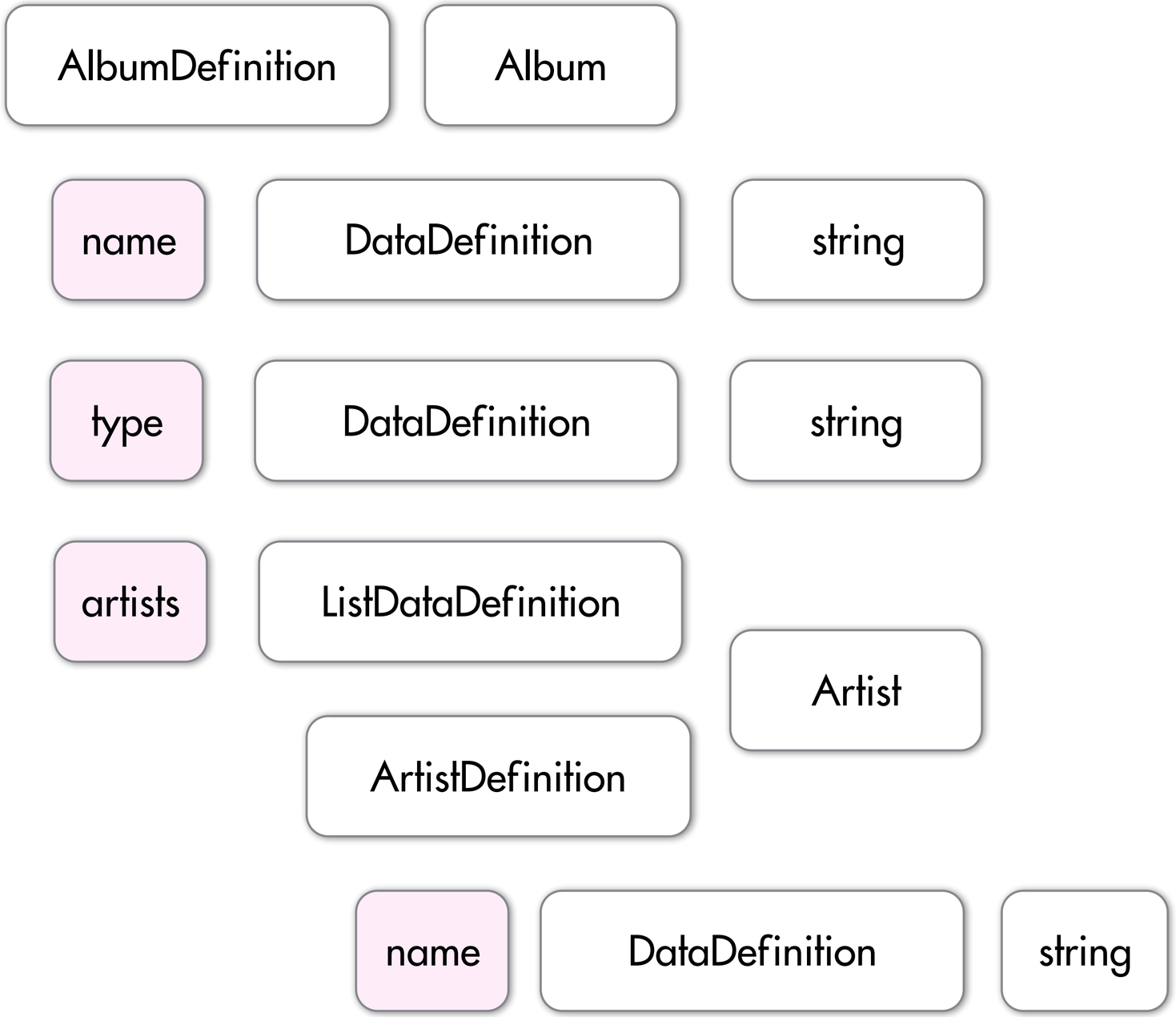
- Data definitions can be lists of data types i.e. an indexed array equivalent.
- This is important for Xero because the API returns a list of items in addition to other odd types like Tracking Categories, Line Items, and Addresses.

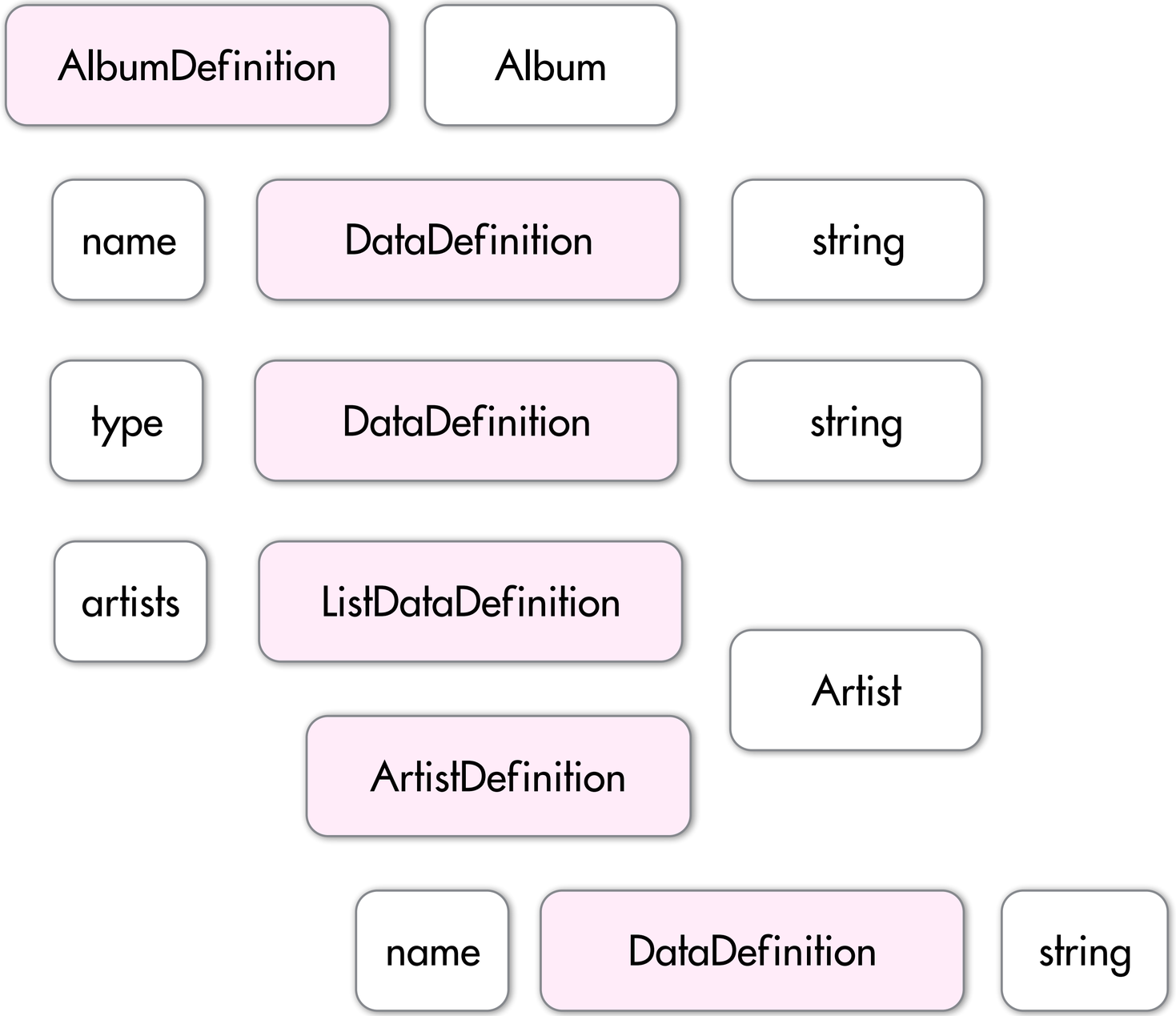
```php
class ExampleDefinition extends ComplexDataDefinitionBase {

  public function getPropertyDefinitions() {
    if (!isset($this->propertyDefinitions)) {
      $info = &$this->propertyDefinitions;
      $info['name'] = DataDefinition::create('string')
        ->setLabel('Name')
        ->setDescription('A string property that is required.')
        ->setRequired(TRUE);

      $info['primary'] = ColorDefinition::create('typed_example_color')
        ->setLabel('Primary Color')
        ->setDescription('A complex data type as a child property.');

      $info['secondary'] = ListDataDefinition::create('typed_example_color')
        ->setLabel('Secondary Colors')
        ->setDescription('A list of complex data types as a child property.');
    }

    return $this->propertyDefinitions;
  }

}
```

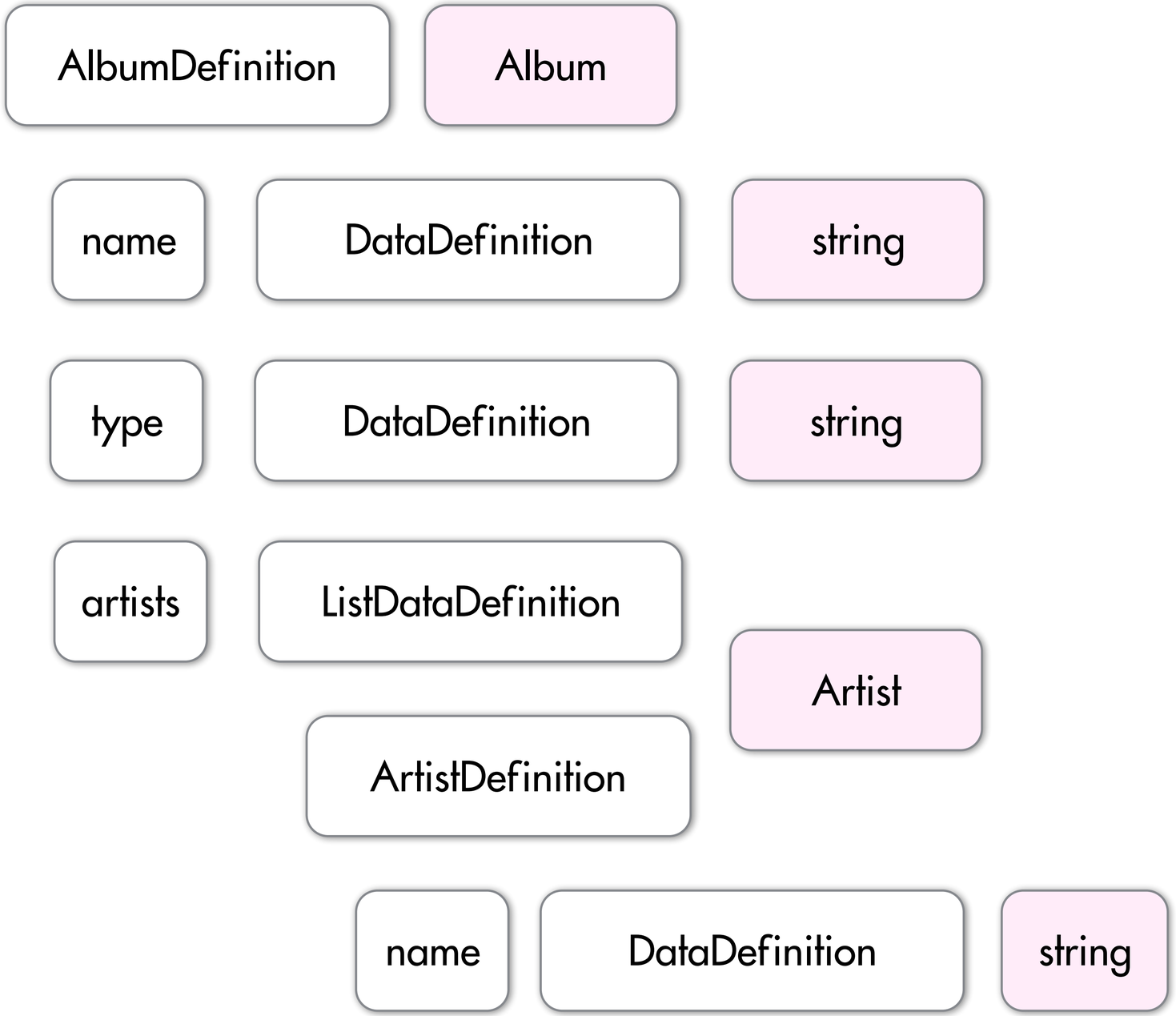
Exercise: Spotify

- Album
  - track[]
  - artist[]
  - type
  - name

https://developer.spotify.com/web-api/object-model/


AlbumDefinition
Album
name
DataDefinition
string
type
DataDefinition
string
artists
ListDataDefinition
Artist
ArtistDefinition
name
DataDefinition
string


AlbumDefinition
Album
name
DataDefinition
string
type
DataDefinition
string
artists
ListDataDefinition
Artist
ArtistDefinition
name
DataDefinition
string


AlbumDefinition
Album
name
DataDefinition
string
type
DataDefinition
string
artists
ListDataDefinition
Artist
ArtistDefinition
name
DataDefinition
string

Making Use of All of This

- Use Typed Data methods on Complex and Primitive data types.
- Using Typed Data Manager to create Typed Data of Data Types defined by Data Definitions.
- Normalization service that helps to transform Xero API into data types and vice versa.
- And use Serializer and Guzzle in a consistent and maintainable way.
- Extending Typed Data beyond its wildest dreams.

The Basics

- All familiar territory for those familiar with Entity API.
  - get/set returns the Typed Data class representation of the data type.
  - getValue/setValue returns the values in native PHP types (array, string)
    - getCastedValue returns typed cast values for integer, float.
  - TypedDataManager creates data from definitions (usually abstracted away).

api.drupal.org. https://api.drupal.org/api/drupal/core!core.api.php/group/typed_data/8.

chx, mradcliffe, dixon_, Berdir. 2015. Typed Data API in Drupal 8. https://www.drupal.org/node/1794140.

Example: Create a Type Data from a Data Definition

typed_example.module

https://www.drupal.org/sandbox/mradcliffe/2805531

```php
function typed_example_example() {
  $typedDataManager = \Drupal::service('typed_data_manager');

  // 1. Create a data definition for a given data type plugin.
  $definition = $typedDataManager->createDataDefinition('typed_example_type');

  // Optionally get some default values to populate.
  $default_values = [
    'name' => 'Red',
    'primary' => ['red' => 200.0, 'green' => 0.0, 'blue' => 0.0],
    'secondary' => [
      ['red' => 0.0, 'green' => 0.0, 'blue' => 200.0],
      ['red' => 200.0, 'green' => 200.0, 'blue' => 0.0],
    ],
  ];

  // 2. Create typed data from the definition and any default values.
  $data = $typedDataManager->create($definition, $default_values);
}
```

```php
function typed_example_example() {
  $typedDataManager = \Drupal::service('typed_data_manager');

  // 1. Create a data definition for a given data type plugin.
  $definition = $typedDataManager->createDataDefinition('typed_example_type');

  // Optionally get some default values to populate.
  $default_values = [
    'name' => 'Red',
    'primary' => ['red' => 200.0, 'green' => 0.0, 'blue' => 0.0],
    'secondary' => [
      ['red' => 0.0, 'green' => 0.0, 'blue' => 200.0],
      ['red' => 200.0, 'green' => 200.0, 'blue' => 0.0],
    ],
  ];

  // 2. Create typed data from the definition and any default values.
  $data = $typedDataManager->create($definition, $default_values);
}
```

```php
function typed_example_example() {
  $typedDataManager = \Drupal::service('typed_data_manager');

  // 1. Create a data definition for a given data type plugin.
  $definition = $typedDataManager->createDataDefinition('typed_example_type');

  // Optionally get some default values to populate.
  $default_values = [
    'name' => 'Red',
    'primary' => ['red' => 200.0, 'green' => 0.0, 'blue' => 0.0],
    'secondary' => [
      ['red' => 0.0, 'green' => 0.0, 'blue' => 200.0],
      ['red' => 200.0, 'green' => 200.0, 'blue' => 0.0],
    ],
  ];

  // 2. Create typed data from the definition and any default values.
  $data = $typedDataManager->create($definition, $default_values);
}
```

```
function typed_example_example() {

  // 3. Access properties via getters.
  $color = $data->get('primary');

  // 4. Access the raw value with getValue.
  $float = $data->get(‘primary')->get->(‘red')getCastedValue();
}
```

```php
function typed_example_example() {

  // 3. Access properties via getters.
  $color = $data->get('primary');

  // 4. Access the raw value with getValue
  $float = $data->get(‘primary')->get->(‘red')getCastedValue();
}
```

```php
function typed_example_entity() {
  $typedDataManager = \Drupal::service('typed_data_manager');

  // 1. Create the data definition for an user entity.
  $definition = $typedDataManager->createDataDefinition('entity:user');

  // 2. Create Typed Data representation of an user entity.
  $userData = $typedDataManager->create($definition);
}
```

Example: XeroQuery used by a Form to display a list of Accounts.

Drupal\xero\Form\DefaultSettingsForm::buildForm

```php
class DefaultSettingsForm extends ConfigFormBase implements ContainerInjectionInterface {

  public function buildForm(array $form, FormStateInterface $form_state) {

    try {
      $accounts = $this->query->getCache('xero_account');

      foreach ($accounts as $account) {
        if ($account->get('Code')->getValue()) {
          $account_options[$account->get('Code')->getValue()] = $account->get('Name')->getValue();
        }
      }
    }
    catch (RequestException $e) {}

    $form['defaults']['account'] = array(
      '#type' => 'select',
      '#title' => $this->t('Default Account'),
      '#description' => $this->t('Choose a default account.'),
      '#options' => $account_options,
      '#default_value' => $config->get('defaults.account'),
    );
```

```
class DefaultSettingsForm extends ConfigFormBase implements ContainerInjectionInterface {

  public function buildForm(array $form, FormStateInterface $form_state) {

    try {
      $accounts = $this->query->getCache('xero_account');

      foreach ($accounts as $account) {
        if ($account->get('Code')->getValue()) {
          $account_options[$account->get('Code')->getValue()] = $account->get('Name')->getValue();
        }
      }
    }
    catch (RequestException $e) {}

    $form['defaults']['account'] = array(
      '#type' => 'select',
      '#title' => $this->t('Default Account'),
      '#description' => $this->t('Choose a default account.'),
      '#options' => $account_options,
      '#default_value' => $config->get('defaults.account'),
    );
```

```php
class DefaultSettingsForm extends ConfigFormBase implements ContainerInjectionInterface {

  public function buildForm(array $form, FormStateInterface $form_state) {

    try {
      $accounts = $this->query->getCache('xero_account');

      foreach ($accounts as $account) {
        if ($account->get('Code')->getValue()) {
          $account_options[$account->get('Code')->getValue()] = $account->get('Name')->getValue();
        }
      }
    }
    catch (RequestException $e) {}

    $form['defaults']['account'] = array(
      '#type' => 'select',
      '#title' => $this->t('Default Account'),
      '#description' => $this->t('Choose a default account.'),
      '#options' => $account_options,
      '#default_value' => $config->get('defaults.account'),
    );
```

(De)normalization

- Normalizer classes transform data into Typed Data and vice versa.
- Normalization is obscured from use because it runs as part of serialization.
- Core offers some standard normalization, but not all 3rd party APIs conform to Drupal.

Define a Normalizer

- Normalization classes must be defined as services in the service container.
  - These are also “tagged” (container compiler pass).
  - Extend Drupal\serialization\Normalizer\ComplexDataNormalizer

xero.services.yml

```
services:
  xero.normalizer:
    class: Drupal\xero\Normalizer\XeroNormalizer
    arguments: ['@typed_data_manager']
    tags:
      - { name: normalizer }
```

```php
class XeroNormalizer extends ComplexDataNormalizer implements DenormalizerInterface {

  protected $supportedInterfaceOrClass = 'Drupal\xero\TypedData\XeroTypeInterface';

  public function __construct(TypedDataManager $typed_data_manager) {
    $this->typedDataManager = $typed_data_manager;
  }

  public function normalize($object, $format = NULL, array $context = array()) {
    // Get the array map.
    $items = array();

    $items[$object->getName()] = parent::normalize($object, $format, $context);

    return $items;
  }

  // Snip...

}
```

```php
class XeroNormalizer extends ComplexDataNormalizer implements DenormalizerInterface {

  public function denormalize($data, $class, $format = NULL, array $context = array()) {
    // Snip...

    $name = $class::$xero_name;
    $plural_name = $class::$plural_name;

    // Wrap the data an array if there is a single object returned.
    if (count(array_filter(array_keys($data[$plural_name][$name]), 'is_string'))) {
      $data[$plural_name][$name] = array($data[$plural_name][$name]);
    }

    $list_definition = $this->typedDataManager
      ->createListDataDefinition($context['plugin_id']);
    $items = $this->typedDataManager
      ->create($list_definition, $data[$plural_name][$name]);

    return $items;
  }
}
```

Benjamin Eberlei. 2013.01.18. symfony/symfony#9ff8dfdc50e8e1845a3f5559ab4e646f46bbff37.

Use Guzzle and Serializer to Populate Typed Data

10:45 Thur: https://events.drupal.org/dublin2016/sessions/guzzle-extraordinary-http-client

```php
class XeroQuery /*implements XeroQueryInterface */ {

  public function execute() {
    try {
      // Snip...
      $context = [
        'xml_root_node_name' => $endpoint,
        'plugin_id' => $this->type,
      ];

      $this->options['body'] = $this->serializer->serialize($this->data, $this->format, $context);

      /** @var \Psr\Http\Message\ResponseInterface $response */
      $response = $this->client->{$this->method}($endpoint, $this->options);

      /** @var \Drupal\xero\TypedData\XeroTypeInterface $data */
      $data = $this->serializer->deserialize(
        $response->getBody()->getContents(),
        $data_class,
        $this->format,
        $context);
      return $data;
    }
    catch (RequestException $e) {}
  }
```

```php
class XeroQuery /*implements XeroQueryInterface */ {

  public function execute() {
    try {
      // Snip...
      $context = [
        'xml_root_node_name' => $endpoint,
        'plugin_id' => $this->type,
      ];

      $this->options['body'] = $this->serializer->serialize($this->data, $this->format, $context);

      /** @var \Psr\Http\Message\ResponseInterface $response */
      $response = $this->client->{$this->method}($endpoint, $this->options);

      /** @var \Drupal\xero\TypedData\XeroTypeInterface $data */
      $data = $this->serializer->deserialize(
        $response->getBody()->getContents(),
        $data_class,
        $this->format,
        $context);
      return $data;
    }
    catch (RequestException $e) {}
  }
```

```php
class XeroQuery /*implements XeroQueryInterface */ {

  public function execute() {
    try {
      // Snip...
      $context = [
        'xml_root_node_name' => $endpoint,
        'plugin_id' => $this->type,
      ];

      $this->options['body'] = $this->serializer->serialize($this->data, $this->format, $context);

      /** @var \Psr\Http\Message\ResponseInterface $response */
      $response = $this->client->{$this->method}($endpoint, $this->options);

      /** @var \Drupal\xero\TypedData\XeroTypeInterface $data */
      $data = $this->serializer->deserialize(
        $response->getBody()->getContents(),
        $data_class,
        $this->format,
        $context);
      return $data;
    }
    catch (RequestException $e) {}
  }
```

```php
class XeroQuery /*implements XeroQueryInterface */ {

  public function execute() {
    try {
      // Snip...
      $context = [
        'xml_root_node_name' => $endpoint,
        'plugin_id' => $this->type,
      ];

      $this->options['body'] = $this->serializer->serialize($this->data, $this->format, $context);

      /** @var \Psr\Http\Message\ResponseInterface $response */
      $response = $this->client->{$this->method}($endpoint, $this->options);

      /** @var \Drupal\xero\TypedData\XeroTypeInterface $data */
      $data = $this->serializer->deserialize(
        $response->getBody()->getContents(),
        $data_class,
        $this->format,
        $context);
      return $data;
    }
    catch (RequestException $e) {}
  }
```

Hello, Typed Element Builder

- Typed Widget module provides a service to dynamically create forms from data definition.
  - Supports entities and fields.
  - Main use is for other complex and primitive data types.
- Started from similar code in my xero module, and I hope it is of use to those using Typed Data API.
- Form state values are structured to be passed back into creating the respective data type.

Typed Element Builder Usage

- Get an element for a normal Typed Data data type.
  - `getElementFor('datetime_iso8601')`
- Get an element for an entity, or a field of an entity.
  - `getElementFor('entity:user')`
  - `getElementFor('entity:user', 'mail')`
- Get an element for a field item definition without the context an entity.
  - `getElementFor('field_item:telephone')`

```php
public function buildForm(array $form, FormStateInterface $form_state) {

  $form['actions'] = ['#type' => 'actions'];
  $form['actions']['submit'] = [
    '#type' => 'submit',
    '#value' => $this->t('Submit'),
  ];

  return $form;
}

/**
 * {@inheritdoc}
 */
public function submitForm(array &$form, FormStateInterface $form_state) {

}

/**
 * {@inheritdoc}
 */
static public function create(ContainerInterface $container) {
  return new static(
    $container->get('typed_widget.element_builder'),
    $container->get('typed_data_manager')
  );
```

Example Color Form

Name *

A string property that is required.

Red *

Green *

Blue *

Typed Widget.Next

- In infancy (i.e. ugly code) and need help to make it more extendable.
- Other ideas to improve Typed Data usability:
  - Provide a non-entity complex data type interface with useful methods such as buildElement and viewElement.
- https://drupal.org/project/typed_widget

Typed Data and PHPUnit Summary

- Unit tests are awesome and fast, but Typed Data Manager is big and scary. :(
- Typed Data calls itself so mocking is not straight-forward. :(
- Data definitions use static methods, which is not great for isolating unit tests. :(
- Need to mock the service container. :(

PHPUnit

- On the other hand, I like seeing pretty coverage and complexity graphs. :-)

- But what is actually necessary for decent code coverage and analysis?

17:00 Today: https://events.drupal.org/dublin2016/sessions/automated-testing-phpunit-all-way

10:45 Tomorrow: https://events.drupal.org/dublin2016/sessions/what-type-testing-good-me

Unit Tests for Data Definitions

- DataDefinitionInterface::getPropertyDefinitions
  - Easy to get coverage for this method as it's only returning an array of more data definitions.
  - Coverage and testing here does not provide much value other than making a pretty green color in my coverage report.
  - Data definitions are created via static methods.
  - Entity and Field definitions are much more difficult to test as those specific classes rely on the service container.

```php
namespace Drupal\Tests\typed_example\Unit\TypedData;

use Drupal\Tests\UnitTestCase;
use Drupal\typed_example\TypedData\ColorDefinition;

/**
 * Test that the property definitions exist for Color Definition.
 *
 * @group typed_example
 */
class ColorDefinitionTest extends UnitTestCase {

  public function testGetPropertyDefinitions() {
    $definition = new ColorDefinition();
    $properties = $definition->getPropertyDefinitions();

    $this->assertEquals('Red', $properties['red']->getLabel());
    $this->assertEquals('Green', $properties['green']->getLabel());
    $this->assertEquals('Blue', $properties['blue']->getLabel());
  }

}
```

Welcome to Code Smell: Typed Data Manager

- The static create method in complex data definitions will call TypedDataManager through \Drupal:
    - Mock objects must carefully re-construct the order of how create will be called on for every definition in a complex data definition. Depending on consecutive calls is a test anti-pattern.
        - Or be lazy and a little sloppy if order isn't important?
- The getPropertyInstance method is the main method that I mock to on child properties of a complex data type.

With Prophecy

ExampleTest (typed_example)

```php
protected function setUp() {

  $definition = ExampleDefinition::create('typed_example_type');
  $definition->setClass('\Drupal\typed_example\Plugin\DataType\Example');

  // Snip...
  $this->example = new Example($definition);

  $stringDefinition = DataDefinition::create('string');
  $stringDefinition->setClass('\Drupal\Core\TypedData\Plugin\DataType\StringData');
  $floatDefinition = DataDefinition::create('float');
  $floatDefinition->setClass('\Drupal\Core\TypedData\Plugin\DataType\FloatData');
  $floatData = new FloatData($floatDefinition);
  $floatData->setValue(100);
  $stringData = new StringData($stringDefinition);
  $stringData->setValue('Test example');

  $colorProphecy = $this->prophesize('\Drupal\typed_example\Plugin\DataType\Color');
  $colorProphecy->get('red')->willReturn($floatData);
  $color = $colorProphecy->reveal();

  $listProphecy = $this->prophesize('\Drupal\typed_example\Plugin\DataType\ExampleColorItemList');
  $listProphecy->get(0)->willReturn($color);
  $list = $listProphecy->reveal();
```

```
  // Snip...
  // Mock Typed Data Manager.
  $typedDataProphecy = $this->prophesize('\Drupal\Core\TypedData\TypedDataManagerInterface');
  $typedDataProphecy->getPropertyInstance(Argument::any(), 'primary', Argument::any())
    ->willReturn($color);
  $typedDataProphecy->getPropertyInstance(Argument::any(), 'secondary', Argument::any())
    ->willReturn($list);
  $typedDataProphecy->getPropertyInstance(Argument::any(), 'red', Argument::any())
    ->willReturn($floatData);
  $typedDataProphecy->getPropertyInstance(Argument::any(), 'name', Argument::any())
    ->willReturn($stringData);

  $container = new ContainerBuilder();
  $container->set('typed_data_manager', $typedDataProphecy->reveal());
  \Drupal::setContainer($container);

  $this->example->setValue([
    'name' => 'Test example',
    'primary' => $color,
    'secondary' => $list,
  ]);
}
```

```php
public function testProperty($class_name, $property_name) {
  $this->assertInstanceOf($class_name, $this->example->get($property_name));
}

public function propertyTestProvider() {
  return [
    ['\Drupal\Core\TypedData\Plugin\DataType\StringData', 'name'],
    ['\Drupal\typed_example\Plugin\DataType\Color', 'primary'],
    ['\Drupal\Core\TypedData\Plugin\DataType\ItemList', 'secondary'],
  ];
}
```

With PHPUnit Mocks

XeroListNormalizerTest (xero)

```
    $this->typedDataManager->expects($this->any())
      ->method('create')
      ->with($this->listDefinition, $this->data['Account'])
      ->will($this->returnValue($itemList));

    $this->typedDataManager->expects($this->any())
      ->method('getPropertyInstance')
      ->withConsecutive(
        array($itemList, 0, $account),
        array($account, 'AccountID', $this->data['Account'][0]['AccountID']),
        array($account, 'Code', $this->data['Account'][0]['Code']),
        array($account, 'Name', $this->data['Account'][0]['Name']),
        array($account, 'Type', $this->data['Account'][0]['Type'])
      )
      ->will($this->onConsecutiveCalls($account, $guid, $code, $name, $type));

    $itemList->setValue([0 => $account]);

    $items = $this->typedDataManager->create($this->listDefinition, $this->data['Account']);

    $data = $this->normalizer->normalize($items, 'xml', array('plugin_id' => 'xero_account'));

    $this->assertArrayEquals($expect, $data, print_r($data, TRUE));
}
```

A poem about Mocking Drupal

What I learn not to do

I resign myself to that fate.

Besides, core does it too.

So please do not hate

this container so small

is not actually in my app at all.

I do not wish to load,

or directly invoke the container,

to get variables into my code.

But to be a better maintainer,

forgive that what I leverage.

All I want is pretty coverage.

Re: Unit Tests, Mocking and Typed Data

"You're crazy!"

-webchick (June 2015)

Why Go Through All of That?

- Typed Data API is difficult to mock because of various code smells, but it is similar to other sub-systems including the Entity API.
  - Previous code was not testable and untyped associative arrays.
    - Ever written an automated test for an external API? In a public repository?

Why Go Through All of That?

- Typed Data API is difficult to mock because of various code smells, but it is similar to other sub-systems including the Entity API.
  - Previous code was not testable and untyped associative arrays.
    - Ever written an automated test for an external API? In a public repository?
- Data definitions and types better model non-Drupal data, which makes it less complex to integrate (no storage, access handler, form complexity, entity normalization).

Why Go Through All of That?

- Typed Data API is difficult to mock because of various code smells, but it is similar to other sub-systems including the Entity API.
    - Previous code was not testable and untyped associative arrays.
        - Ever written an automated test for an external API? In a public repository?
- Data definitions and types better model non-Drupal data, which makes it less complex to integrate (no storage, access handler, form complexity, entity normalization).
- Using composite data types to represent data better integrates with the Drupal core and contrib.

Typed Data Summary

- How to create data types and data definitions.
- Using Typed Data:
  - Instantiating and using data types.
  - With Serializer to integrate to 3rd party API.
  - With Typed Widget to dynamically create forms from data definitions.
- Testing Typed Data with PHPUnit and mocking Typed Data Manager.

What else..?

- The floor is open to ask, discuss or share:
  - Typed Data use cases.
  - Questions about Typed Data Manager, Typed Widget
  - Comments about Typed Data API complexity and improvements.

Matthew Radcliffe
Kosada
mradcliffe
@mattkineme

Evaluate this session: events.drupal.org/dublin2016/schedule

DUBLIN
DRUPALCON

JOIN US FOR CONTRIBUTION SPRINTS

First Time Sprinter Workshop - 9:00-12:00 - Room Wicklow2A

Mentored Core Sprint - 9:00-18:00 - Wicklow Hall 2B

General Sprints - 9:00 - 18:00 - Wicklow Hall 2A