ALEXANDER VARWIJK - 21ST OF SEPTEMBER 2022

BUILDING A GRAPHQL API

Beyond the basics

WHOAM(I)

ALEXANDER VARWIJK

- Lead Front-End Engineer at

- 20 Years of programming
- 10+ years professionally

- Addicted to coffee
- My first DrupalCon was in Prague in 2013!

COMMUNITY ENGAGEMENT PLATFORM AS A SERVICE

open social®

Helping organizations across the globe create communities that drive real-world change

Always looking for Drupal developers

https://careers.getopensocial.com

WHAT ISN'T IN THIS TALK?

✅ In this talk

- Setting up a modular schema
- GraphQL module deep-dive

❌ Not in this talk

- Schema Design Course
- Automated Testing

FOLLOW ALONG OR FIND IT LATER

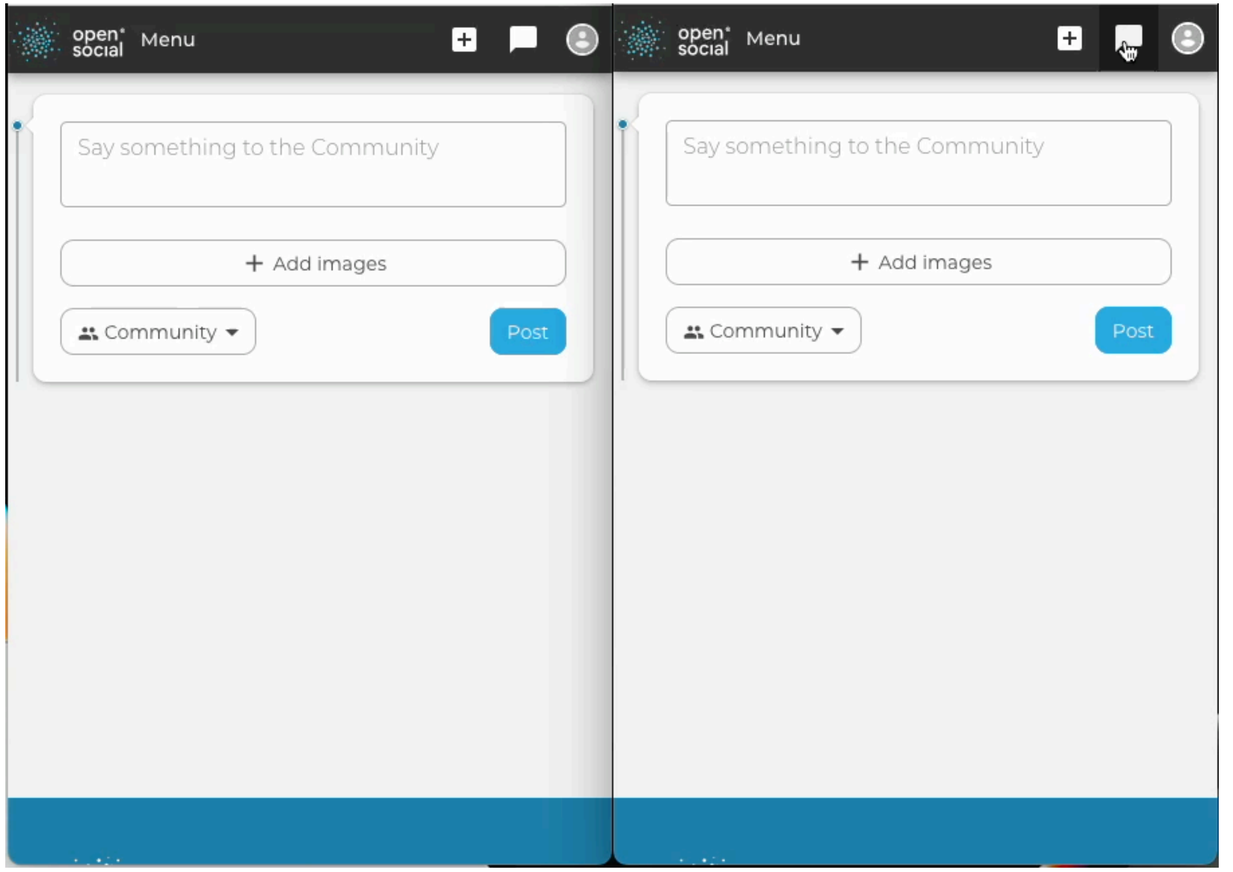
https://www.alexandervarwijk.com/talks/2022-09-21-building-a-graphql-api-beyond-the-basics

BUILDING A REAL-TIME CHAT

SCHEMA DESIGN

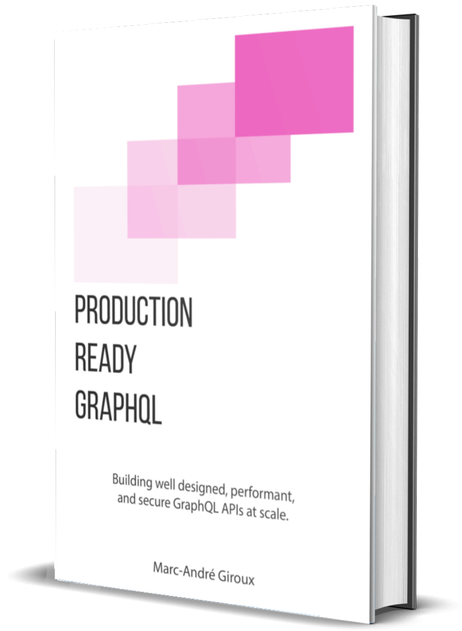
- Defining the goal of what we're building

- As a chat participant I want to be able to send chat messages to and receive chat messages from other participant(s) without refreshing my page so that I can easily communicate with others


open social
Menu
Say something to the Community
Add images
Community
Post
open social
Menu
Say something to the Community
Add images
Community
Post

PRODUCTION READY GRAPHQL

Building well designed, performant, and secure GraphQL APIs at scale.



Every developer at Open Social has access to this book

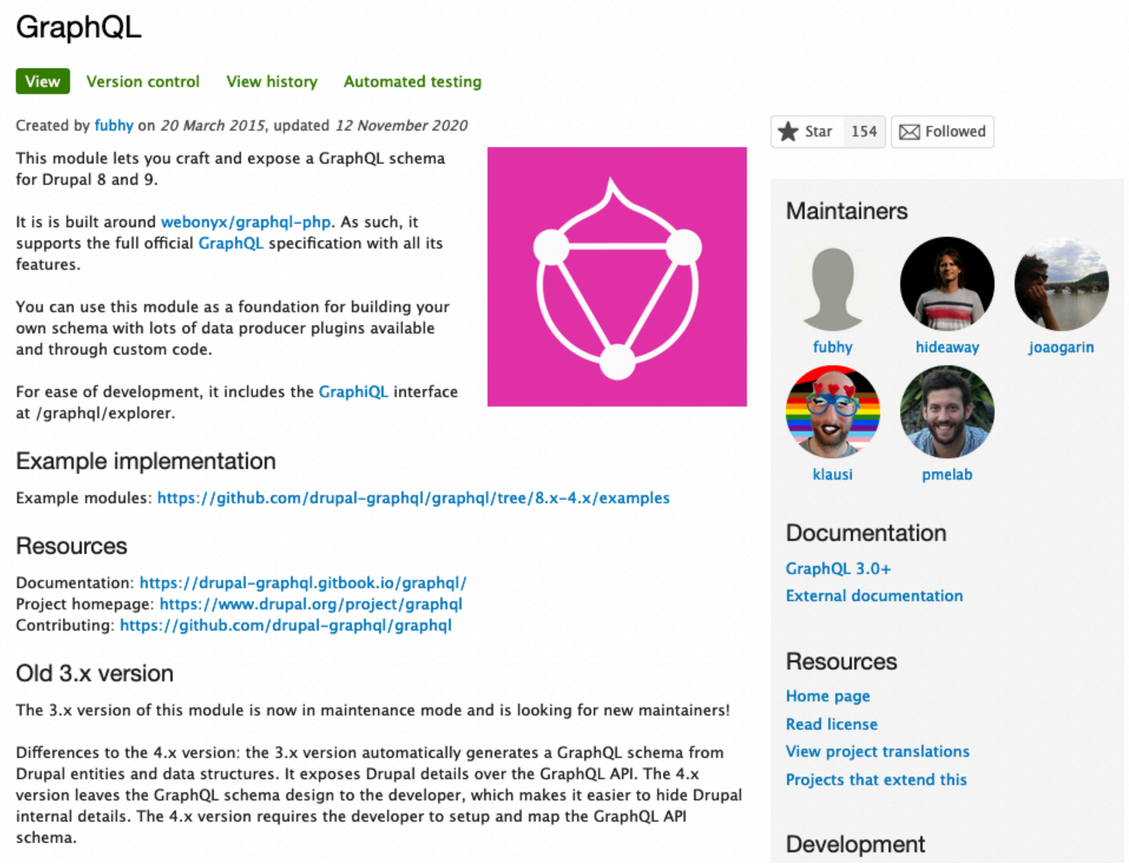
https://book.productionreadygraphql.com/

Written by Marc-Andre Giroux

THE GRAPHQL MODULE

THE GRAPHQL MODULE

Because there's always a module for that ;-)



GraphQL

View | Version control | View history | Automated testing

Created by fubhy on *20 March 2015*, updated *12 November 2020*

This module lets you craft and expose a GraphQL schema for Drupal 8 and 9.

It is is built around webonyx/graphql-php. As such, it supports the full official GraphQL specification with all its features.

You can use this module as a foundation for building your own schema with lots of data producer plugins available and through custom code.

For ease of development, it includes the GraphiQL interface at /graphql/explorer.

Example implementation

Example modules: https://github.com/drupal-graphql/graphql/tree/8.x-4.x/examples

Resources

Documentation: https://drupal-graphql.gitbook.io/graphql/
Project homepage: https://www.drupal.org/project/graphql
Contributing: https://github.com/drupal-graphql/graphql

Old 3.x version

The 3.x version of this module is now in maintenance mode and is looking for new maintainers!

Differences to the 4.x version: the 3.x version automatically generates a GraphQL schema from Drupal entities and data structures. It exposes Drupal details over the GraphQL API. The 4.x version leaves the GraphQL schema design to the developer, which makes it easier to hide Drupal internal details. The 4.x version requires the developer to setup and map the GraphQL API schema.

Star 154 | Followed

Maintainers

fubhy | hideaway | joaogarin | klausi | pmelab

Documentation

GraphQL 3.0+
External documentation

Resources

Home page
Read license
View project translations
Projects that extend this

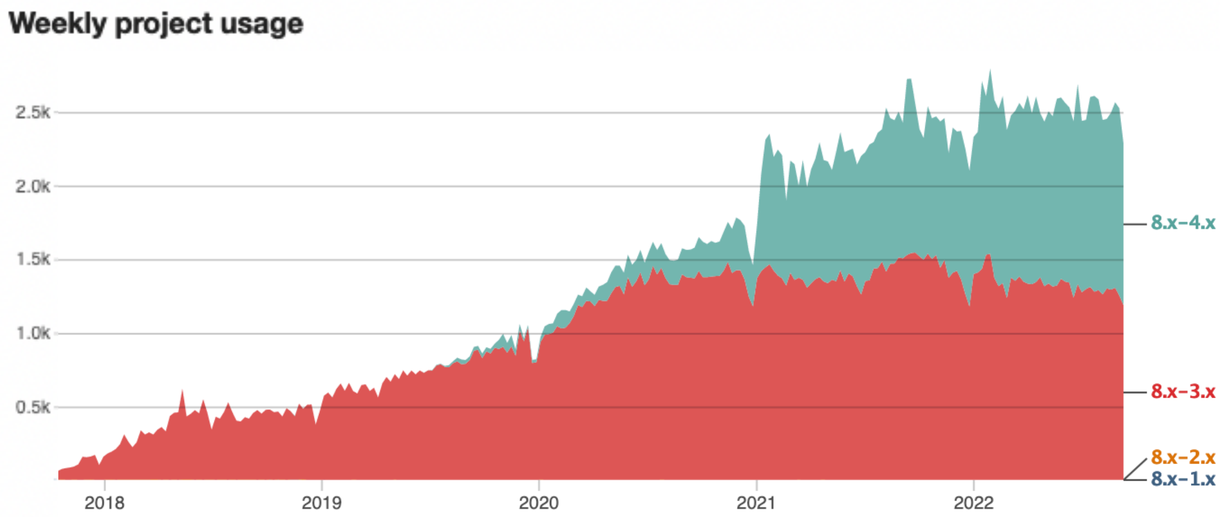
Development

- Uses the webonyx/graphql-php library under the hood
- Contributions and issue tracking happens on GitHub
- 3.x and 4.x are actively used by the community

DRUPAL/GRAPHQL

Two major versions

- Version 3 features automatic schema generation. This leaks Drupal implementation details.
- Version 4 requires writing your own schema. This can result in a cleaner schema but takes more effort.



GRAPHQL_COMPOSE

Automatic schema generation in drupal/graphql v4

GraphQL Compose

View | Version control | View history | Automated testing

This project is not covered by Drupal's security advisory policy.

Star 10 | Followed

> Toolkit for generating GraphQL schemas in Drupal.

The main goal of this project is to provide a GraphQL spec compliant schema generation for the GraphQL module v4.x

There are plans to provide a GUI (YAML/JSON file will be supported for the initial release) to allow users to define how they want to name things and what to expose.

There are no plans to provide CLI commands, since that will probably limit the number of users that can take advantage of this module to only users that feel comfortable using the CLI.

Other possible features from current v3.x modules that could considered to be added:

- GraphQL Mutation
- GraphQL Responsive Image
- GraphQL Metatag
- GraphQL Extras
- GraphQL Formatters
- GraphQL Config
- GraphQL Entity Definitions
- GraphQL Redirect / Redirect Entity
- GraphQL OAuth
- GraphQL JWT
- GraphQL WebP
- GraphQL Views

Current Field type implementation

See spreadsheet here for progress. (need/want access?, ask for it).

Maintainers


jmolivas


juanramonperez

Issues for GraphQL Compose

To avoid duplicates, please search before submitting a new issue.

Search

Advanced search

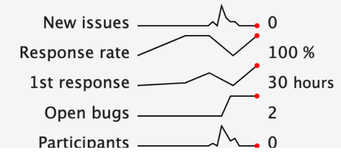
All issues

4 open, 10 total

Bug report

2 open, 6 total

Statistics



- jmolivas has been hard at work rebuilding automatic schema generation on the new drupal/graphql v4 architecture
- Updates in #graphql on the Drupal Slack
- graphql_compose project on Drupal.org

IMPLEMENTING OUR API

DEFINING A BASE SCHEMA

Utilising inheritance and Drupal's modularity

```
/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
 *\
 * )
```

Our base schema helps us with some common types

Schema extensions can re-use those types without having to define them

DEFINING A BASE SCHEMA

modules/custom/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

1. Our schema must be in the Plugin\GraphQL\Schema namespace to be discovered
2. We use the schema annotation to define our base schema
3. Our only required function is getResolverRegistry

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;

/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
class DrupalConBaseSchema
   extends SdlSchemaPluginBase {

  /**
   * {@inheritdoc}
   */
  public function getResolverRegistry() {
    return new ResolverRegistry();
  }

}
```

DEFINING A BASE SCHEMA

modules/custom/drupalcon/graphql/drupalcon.graphqls

Only types which can be shared by all the other parts of our schema

Ideally in your version you include comments ;-)

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

type Query

type Mutation

type Subscription
```

```
type DateTime {
  timestamp: Timestamp!
}

interface Node { id: ID! }

scalar Cursor

interface Connection {
  edges: [Edge!]!
  nodes: [Node!]!
  pageInfo: PageInfo!
}

interface Edge {
  cursor: Cursor!
  node: Node!
}

type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
  startCursor: Cursor
  endCursor: Cursor
}
```

DEFINING A BASE SCHEMA

modules/custom/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\Schema;


use Drupal\graphql\GraphQL\ResolverBuilder;
use Drupal\graphql\GraphQL\ResolverRegistry;

/**
 * @Schema(
 *   id = "drupalcon",
 *   name = "A DrupalCon base GraphQL Schema"
 * )
 */
class DrupalConBaseSchema extends SdlSchemaPluginBase {

  /**
   * {@inheritdoc}
   */
  public function getResolverRegistry() {
    return new ResolverRegistry();
  }

}
```

DEFINING A BASE SCHEMA

modules/custom/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

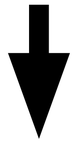
```
public function getResolverRegistry() {
  $registry = new ResolverRegistry();

  return $registry;
}
```

DEFINING A BASE SCHEMA

modules/contrib/graphql/src/GraphQL/ResolverRegistry.php

```
type DateTime {
  timestamp: Timestamp!
}
```

```
/**
 * Contains all the mappings how to resolve a GraphQL request.
 */
class ResolverRegistry implements ResolverRegistryInterface {

  /**
   * {@inheritdoc}
   */
  public function addFieldResolver($type, $field, ResolverInterface $resolver) {
    $this->fieldResolvers[$type][$field] = $resolver;
    return $this;
  }

}
```



DEFINING A BASE SCHEMA

modules/contrib/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```php
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  $registry->addFieldResolver('Connection', 'edges',
    $builder->produce('connection_edges')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'nodes',
    $builder->produce('connection_nodes')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'pageInfo',
    $builder->produce('connection_page_info')
      ->map('connection', $builder->fromParent())
  );

  $registry->addFieldResolver('Edge', 'cursor',
    $builder->produce('edge_cursor')->map('edge', $builder->fromParent())
  );
  $registry->addFieldResolver('Edge', 'node',
    $builder->produce('edge_node')->map('edge', $builder->fromParent())
  );

  return $registry;
}
```

DEFINING A BASE SCHEMA

modules/contrib/drupalcon/src/Plugin/GraphQL/Schema/DrupalConBaseSchema.php

```
public function getResolverRegistry() {
  $registry = new ResolverRegistry();
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('DateTime', 'timestamp', $builder->fromParent());

  $registry->addFieldResolver('Connection', 'edges',
    $builder->produce('connection_edges')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'nodes',
    $builder->produce('connection_nodes')->map('connection', $builder->fromParent())
  );
  $registry->addFieldResolver('Connection', 'pageInfo',
    $builder->produce('connection_page_info')
      ->map('connection', $builder->fromParent())
  );

  $registry->addFieldResolver('Edge', 'cursor',
    $builder->produce('edge_cursor')->map('edge', $builder->fromParent())
  );
  $registry->addFieldResolver('Edge', 'node',
    $builder->produce('edge_node')->map('edge', $builder->fromParent())
  );

  return $registry;
}
```

DEFINING A BASE SCHEMA

modules/contrib/drupalcon/src/Plugin/GraphQL/DataProducer/Connection/ConnectionEdges.php

```php
namespace Drupal\drupalcon\Plugin\GraphQL\DataProducer\Connection;

use Drupal\graphql\Plugin\DataProducerPluginCachingInterface;
use Drupal\graphql\Plugin\GraphQL\DataProducer\DataProducerPluginBase;
use Drupal\drupalcon\GraphQL\ConnectionInterface;

/**
 * Produces the edges from a connection object.
 *
 * @DataProducer(
 *   id = "connection_edges",
 *   name = @Translation("Connection edges"),
 *   description = @Translation("Returns the edges of a connection."),
 *   produces = @ContextDefinition("any",
 *     label = @Translation("Edges")
 *   ),
 *   consumes = {
 *     "connection" = @ContextDefinition("any",
 *       label = @Translation("QueryConnection")
 *     )
 *   }
 * )
 */
class ConnectionEdges extends DataProducerPluginBase
  implements DataProducerPluginCachingInterface {}
```

DEFINING A BASE SCHEMA

modules/contrib/drupalcon/src/Plugin/GraphQL/DataProducer/Connection/ConnectionEdges.php

```php
class ConnectionEdges extends DataProducerPluginBase
  implements DataProducerPluginCachingInterface {

  /**
   * Resolves the request.
   *
   * @param \Drupal\social_graphql\GraphQL\ConnectionInterface $connection
   *   The connection to return the edges from.
   *
   * @return mixed
   *   The edges for the connection.
   */
  public function resolve(ConnectionInterface $connection) {
    return $connection->edges();
  }

}
```

DEFINING OUR SCHEMA EXTENSION

Utilising inheritance and Drupal's modularity

```
/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
```

Our schema extension implements a specialised feature in our API

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

1. Our schema must be in the Plugin\GraphQL\SchemaExtension namespace to be discovered
2. We use the SchemaExtension annotation to define our schema extension
3. Our only required function is registerResolvers

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;

use Drupal\graphql\GraphQL\ResolverBuilder;
use Drupal\graphql\GraphQL\ResolverRegistryInterface;
use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {

  /**
   * {@inheritdoc}
   */
  public function registerResolvers(ResolverRegistryInterface $registry) : void {
    $builder = new ResolverBuilder();

    /* Chat Resolvers */
  }

}
```

DEFINING OUR SCHEMA EXTENSION BASE SCHEMA

modules/contrib/drupalcon_chat/graphql/drupalcon_chat.base.graphqls

Defines new types for the chat functionality

```
type ChatMessage implements Node {
    id: ID!
    sent: DateTime!
    sender: Actor
    content: ChatMessageContent!
}

union ChatMessageContent =
    | MediaChatMessageContent
    | UserEventChatMessageContent
    | DeletedChatMessageContent

type MediaChatMessageContent {
    text: String
}

type UserEventChatMessageContent {
    eventType: ChatUserEventType!
    subject: User!
}

enum ChatUserEventType {
    CONVERSATION_CREATED
    JOIN
    PART
}
```

```
type DeletedChatMessageContent {
    _: Boolean
}

type ChatMessageConnection
implements Connection {
    pageInfo: PageInfo!
    edges: [ChatMessageEdge!]!
    nodes: [ChatMessage!]!
}

type ChatMessageEdge implements Edge {
    cursor: Cursor!
    node: ChatMessage!
}

input ViewerSendUserChatMessageInput {
    conversation: ID!
    contents:
MediaChatMessageContentInput!
}

input MediaChatMessageContentInput {
    text: String!
}

type ViewerSendUserChatMessagePayload {
    errors: [Violation!]
    message: ChatMessage
}
```

DEFINING OUR SCHEMA EXTENSION EXTENSIONS

modules/contrib/drupalcon_chat/graphql/drupalcon_chat.extension.graphqls

Defines type extensions for the chat functionality

```
extend type Query {
    chatMessages(
        first: Int, after: Cursor,
        last: Int, before: Cursor,
        reverse: Boolean = false
    ) : ChatMessageConnection!
    chatMessage(id: ID!) : ChatMessage
}

extend type Mutation {
    viewerSendUserChatMessage(
        input: ViewerSendUserChatMessageInput!
    ) : ViewerSendUserChatMessagePayload
}

extend type Subscription {
    chatMessageReceived : ChatMessage!
}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
namespace Drupal\drupalcon_chat\Plugin\GraphQL\SchemaExtension;

use Drupal\graphql\GraphQL\ResolverBuilder;
use Drupal\graphql\GraphQL\ResolverRegistryInterface;
use Drupal\graphql\Plugin\GraphQL\SchemaExtension\SdlSchemaExtensionPluginBase;

/**
 * @SchemaExtension(
 *   id = "drupalcon_chat",
 *   name = "DrupalCon - Chat",
 *   description = "Extend GraphQL schema with chat functionality.",
 *   schema = "drupalcon"
 * )
 */
class ChatSchemaExtension extends SdlSchemaExtensionPluginBase {

  /**
   * {@inheritdoc}
   */
  public function registerResolvers(ResolverRegistryInterface $registry) : void {
    $builder = new ResolverBuilder();

    /* Chat Resolvers */
  }

}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();

  $registry->addFieldResolver('Query', 'chatMessage',
    $builder->produce('entity_load_by_uuid')
      ->map('type', $builder->fromValue('chat_message'))
      ->map('uuid', $builder->fromArgument('id'))
  );

}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addFieldResolver('Query', 'chatMessages',
    $builder->produce('messages')
      ->map('after', $builder->fromArgument('after'))
      ->map('before', $builder->fromArgument('before'))
      ->map('first', $builder->fromArgument('first'))
      ->map('last', $builder->fromArgument('last'))
      ->map('reverse', $builder->fromArgument('reverse'))
  );









}
```

THE MESSAGES DATAPRODUCER

Implementation on GitHub in the Drupal\drupalcon_chat\Plugin\GraphQL\DataProducer namespace

DataProducer's with pagination return a EntityConnection instance.

EntityConnection contains the pagination logic and takes a ConnectionQueryHelperInterface as only argument. The implementation of the ConnectionQueryHelperInterface contains:

- The creation of an EntityQuery
- Cursor to object hydration
- Id field configuration
- Sort field configuration
- Creation of the loader promise and object to cursor transformation

See for example Drupal\drupalcon_user\Plugin\GraphQL\QueryHelper\UserQueryHelper

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();









  $registry->addFieldResolver('DeletedChatMessageContent', '_',
    $builder->fromValue(NULL)
  );









}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();

  $registry->addTypeResolver(
    'ChatMessageContent',
    [ChatTypeResolver::class, 'resolveMessageContentType']
  );

}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/ChatTypeResolver.php

```php
/**
 * Type resolver functions for GraphQL types in drupalcon_chat.
 */
class ChatTypeResolver {
  /**
   * Find the concrete type for a MessageContent interface implementation.
   *
   * @param \Drupal\drupalcon_chat\ChatMessageContentInterface $message_content
   *   The message content that should be mapped to a concrete type.
   *
   * @return string
   *   The concrete GraphQL type name.
   */
  public static function resolveMessageContentType(ChatMessageContentInterface $message_content) : string {
    switch ($message_content->getType()) {
      case 'deleted':
        return 'DeletedChatMessageContent';
      case 'media':
        return 'MediaChatMessageContent';
      case 'user_event':
        return 'UserEventChatMessageContent';
      default:
        throw new \RuntimeException("Can not map type '{$message_content->getType()}' to GraphQL Schema");
    }
  }
}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();


  $this->registerMutationResolver(
    $registry,
    $builder,
    'viewerSendUserChatMessage'
  );









}
```

DEFINING OUR SCHEMA EXTENSION

modules/contrib/drupalcon_chat/src/Plugin/GraphQL/SchemaExtension/ChatSchemaExtension.php

```php
public function registerResolvers(ResolverRegistryInterface $registry) : void {
  $builder = new ResolverBuilder();


  $this->registerMutationResolver(
    $registry,
    $builder,
    'viewerSendUserChatMessage'
  );



  // Equivalent to the following under the hood.
  $field_name = camelCaseTOsnake_case($fieldName);
  $registry->addFieldResolver('Mutation', $fieldName,
    $builder->compose(
      $builder->produce($field_name . "_input")
        ->map('input', $builder->fromArgument('input')),
      $builder->produce($field_name)
        ->map('input', $builder->fromParent())
    )
  );


}
```

ADDING A SUBSCRIPTION SERVER

SUBSCRIPTION HANDLING

Real-Time Drupal!

https://www.alexandervarwijk.com/talks/2021-12-10-serving-graphql-subscriptions-using-php-and-drupal

SUBSCRIPTION HANDLING

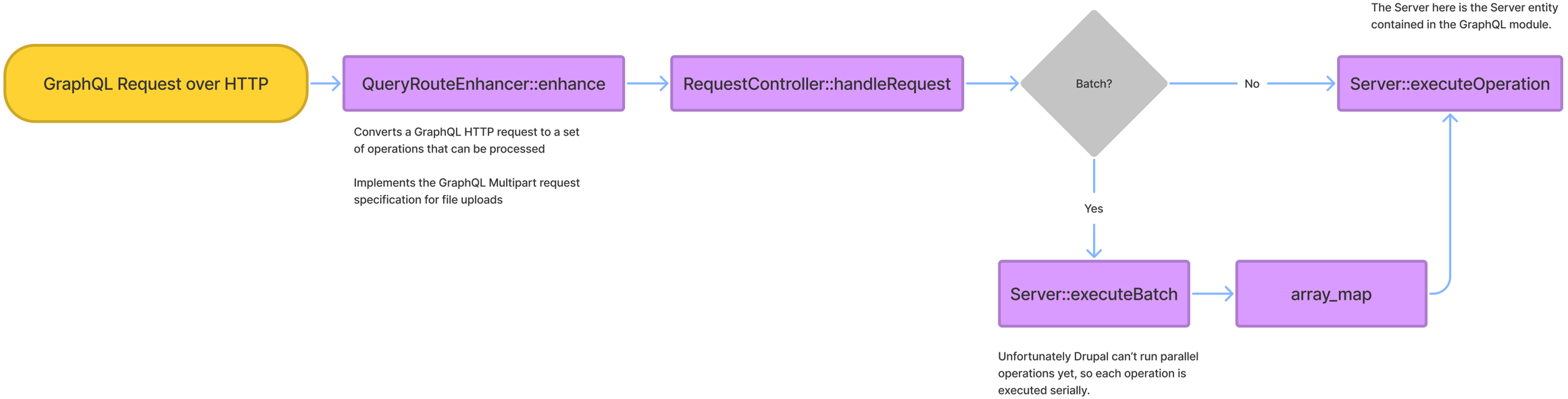
```
type Query {
    chatMessage(id: ID!) : ChatMessage
}

type Subscription {
    chatMessageReceived : ChatMessage!
}
```

SUBSCRIPTION HANDLING

```
$reference_query = Parser::parse(<<<GRAPHQL
   query ChatMessage(\$id: ID!) {
    chatMessage(id: \$id) { id }
  }
GRAPHQL);

$query = Printer::doPrint(
  $this->transformSubscriptionForConcreteType(
    $subscription->document(),
    $reference_query,
    $subscription->operationName()
  )
);

// Use the GraphQL client created for the subscription to ask Open
// Social for the fields needed.
/** @var \OpenSocial\RealTime\GraphQL\GraphQLClient $graphql */
$graphql = $subscription->client()->GraphQL;

/** @var \React\Promise\ExtendedPromiseInterface $promise */
$promise = $graphql->execute($query, $subscription->operationName(), ['id' => $data['message']]);
// Pass the returned data on to the subscription.
$promise->done(
  fn (ExecutionResult $result) => $subscription->client()->send(
    json_encode((new NextMessage($subscription_id, $result))->jsonSerialize())
  )
);
```

INSIDE THE GRAPHQL MODULE

THE JOURNEY OF A GRAPHQL REQUEST



THE JOURNEY OF A GRAPHQL REQUEST

1. Store previous executor
2. Use our own executor (that's what interfaces with Drupal)
3. Get GraphQL configuration
4. Use GraphQL Library Helper to execute operation
5. Ensure Drupal always has cache information for the result
6. Restore previous executor

```php
public function executeOperation(OperationParams $operation) {
  $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
    'create',
  ]);

  try {
    $config = $this->configuration();
    $result = (new Helper())->executeOperation($config, $operation);

    // In case execution fails before the execution stage, we have to wrap the
    // result object here.
    if (!($result instanceof CacheableExecutionResult)) {
      $result = new CacheableExecutionResult(
         $result->data, $result->errors, $result->extensions
      );
      $result->mergeCacheMaxAge(0);
    }
  }
  finally {
    Executor::setImplementationFactory($previous);
  }

  return $result;
}
```

CONFIGURING THE GRAPHQL PHP LIBRARY

```php
public function configuration() {
  $params = \Drupal::getContainer()->getParameter('graphql.config');
  /** @var \Drupal\graphql\Plugin\SchemaPluginManager $manager */
  $manager = \Drupal::service('plugin.manager.graphql.schema');
  $schema = $this->get('schema');

  /** @var \Drupal\graphql\Plugin\SchemaPluginInterface $plugin */
  $plugin = $manager->createInstance($schema);
  if ($plugin instanceof ConfigurableInterface && $config = $this->get('schema_configuration')) {
    $plugin->setConfiguration($config[$schema] ?? []);
  }

  // Create the server config.
  $registry = $plugin->getResolverRegistry();
  $server = ServerConfig::create();
  $server->setDebugFlag($this->get('debug_flag'));
  $server->setQueryBatching(!!$this->get('batching'));
  $server->setValidationRules($this->getValidationRules());
  $server->setPersistentQueryLoader($this->getPersistedQueryLoader());
  $server->setSchema($plugin->getSchema($registry));
  $server->setPromiseAdapter(new SyncPromiseAdapter());
  $server->setContext($this->getContext($plugin, $params));
  $server->setFieldResolver($this->getFieldResolver($registry));

  return $server;
}
```

THE JOURNEY OF A GRAPHQL REQUEST

```php
public function executeOperation(OperationParams $operation) {
  $previous = Executor::getImplementationFactory();
  Executor::setImplementationFactory([
    \Drupal::service('graphql.executor'),
    'create',
  ]);

  try {
    $config = $this->configuration();
    $result = (new Helper())->executeOperation($config, $operation);

    // In case execution fails before the execution stage, we have to wrap the
    // result object here.
    if (!($result instanceof CacheableExecutionResult)) {
      $result = new CacheableExecutionResult(
         $result->data, $result->errors, $result->extensions
      );
      $result->mergeCacheMaxAge(0);
    }
  }
  finally {
    Executor::setImplementationFactory($previous);
  }

  return $result;
}
```

THE JOURNEY OF A GRAPHQL REQUEST

- Ensures the server has a schema
- Ensures the server supports query batching if this is a batched request
- Validates the parameters for the operation
- Loads persisted query if queryId is provided
- Determines the operation to execute
- Ensures HTTP GET is only used for Query and no other operation types

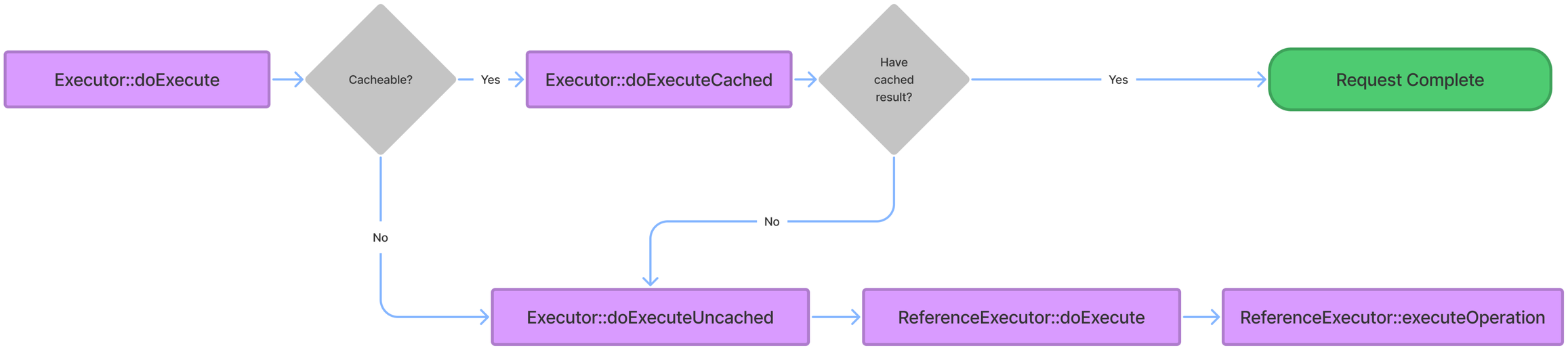
Applies error handling to the result if needed

- Applies our configured operation validation rules. Returns an error if any validation rules are broken.

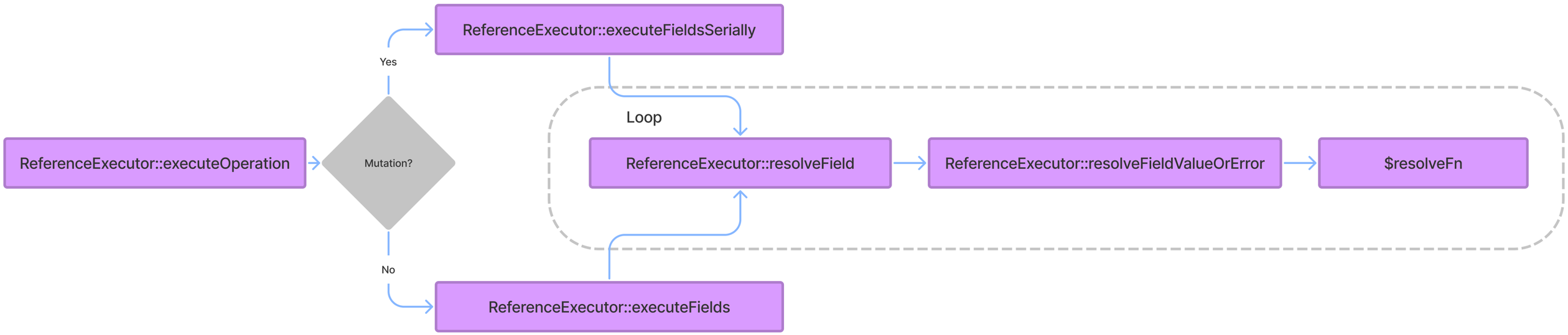
- This happens in the GraphQL library
- Creates an instance of the configured executor

- This is now the Executor of the GraphQL module

THE JOURNEY OF A GRAPHQL REQUEST



THE JOURNEY OF A GRAPHQL REQUEST



THE JOURNEY OF A GRAPHQL REQUEST

```php
/**
 * Returns the default field resolver.
 *
 * @todo Handle this through configuration on the server.
 *
 * Fields that don't explicitly declare a field resolver will use this one
 * as a fallback.
 *
 * @param \Drupal\graphql\GraphQL\ResolverRegistryInterface $registry
 *   The resolver registry.
 *
 * @return null|callable
 *   The default field resolver.
 */
protected function getFieldResolver(ResolverRegistryInterface $registry) {
  return function ($value, $args, ResolveContext $context, ResolveInfo $info) use ($registry) {
    $field = new FieldContext($context, $info);
    $result = $registry->resolveField($value, $args, $context, $info, $field);
    return DeferredUtility::applyFinally($result, function ($result) use ($field, $context) {
      if ($result instanceof CacheableDependencyInterface) {
        $field->addCacheableDependency($result);
      }

      $context->addCacheableDependency($field);
    });
  };
}
```

THE JOURNEY OF A GRAPHQL REQUEST

$value The value of the parent type for the field we're resolving. For root fields this is rootValue from our ServerConfig

$args The arguments provided to the field in the query

$context The context we set in our ServerConfig. A callable that returns a new ResolveConfig instance to keep track of cache metadata

$info Information that may be useful to the field resolver: the field definition, the field name, the expected return type, the nodes in the query referencing the field, the parent type, the path to this field from the root value, the schema used for execution, all fragments in the query, the root value for query execution, the AST of the operation, and variables passed to the query execution

```
        function ($value, $args, ResolveContext $context, ResolveInfo $info) use ($registry) {
  $field = new FieldContext($context, $info);
  $result = $registry->resolveField($value, $args, $context, $info, $field);
  return DeferredUtility::applyFinally($result, function ($result) use ($field, $context) {
    if ($result instanceof CacheableDependencyInterface) {
      $field->addCacheableDependency($result);
    }

    $context->addCacheableDependency($field);
  });
}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

```php
public function resolveField($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  // First, check if there is a resolver registered for this field.
  if ($resolver = $this->getRuntimeFieldResolver($value, $args, $context, $info)) {
    if (!$resolver instanceof ResolverInterface) {
      throw new \LogicException(sprintf('Field resolver for field %s on type %s is not callable.',
         $info->fieldName, $info->parentType->name
      ));
    }

    return $resolver->resolve($value, $args, $context, $info, $field);
  }

  return call_user_func($this->defaultFieldResolver, $value, $args, $context, $info, $field);
}
```

```php
function getRuntimeFieldResolver($value, $args, ResolveContext $context, ResolveInfo $info) {
  return $this->getFieldResolverWithInheritance($info->parentType, $info->fieldName);
}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

```php
public function getFieldResolverWithInheritance(Type $type, string $fieldName) : ?ResolverInterface {
  if ($resolver = $this->getFieldResolver($type->name, $fieldName)) {
    return $resolver;
  }

  if (!$type instanceof ImplementingType) {
    return NULL;
  }

  // Go through the interfaces implemented for the type on which this field is
  // resolved and check if they lead to a field resolution.
  foreach ($type->getInterfaces() as $interface) {
    if ($resolver = $this->getFieldResolverWithInheritance($interface, $fieldName)) {
      return $resolver;
    }
  }

  return NULL;
}
```

```php
public function getFieldResolver($type, $field) {
  return $this->fieldResolvers[$type][$field] ?? NULL;
}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

```php
public function resolveField($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  // First, check if there is a resolver registered for this field.
  if ($resolver = $this->getRuntimeFieldResolver($value, $args, $context, $info)) {
    if (!$resolver instanceof ResolverInterface) {
      throw new \LogicException(sprintf('Field resolver for field %s on type %s is not callable.',
         $info->fieldName, $info->parentType->name
      ));
    }

    return $resolver->resolve($value, $args, $context, $info, $field);
  }

  return call_user_func($this->defaultFieldResolver, $value, $args, $context, $info, $field);
}
```

```php
function getRuntimeFieldResolver($value, $args, ResolveContext $context, ResolveInfo $info) {
  return $this->getFieldResolverWithInheritance($info->parentType, $info->fieldName);
}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

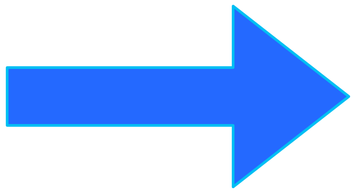
```php
/**
 * Resolves by an argument name lookup.
 */
class Argument implements ResolverInterface {

  protected string $name;

  /**
   * @param string $name
   *   Name of the argument.
   */
  public function __construct($name) {
    $this->name = $name;
  }

  /**
   * {@inheritdoc}
   */
  public function resolve($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
    return $args[$this->name] ?? NULL;
  }

}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

```php
public function resolve($value, $args, ResolveContext $context, ResolveInfo $info, FieldContext $field) {
  $plugin = $this->prepare($value, $args, $context, $info, $field);

  return DeferredUtility::returnFinally(
    $plugin,
    function (DataProducerPluginInterface $plugin) use ($context, $field) {
      foreach ($plugin->getContexts() as $item) {
        /** @var \Drupal\Core\Plugin\Context\Context $item */
        if ($item->getContextDefinition()->isRequired() && !$item->hasContextValue()) {
          return NULL;
        }
      }

      if ($this->cached && $plugin instanceof DataProducerPluginCachingInterface) {
        if (!!$context->getServer()->get('caching')) {
          return $this->resolveCached($plugin, $context, $field);
        }
      }

      return $this->resolveUncached($plugin, $context, $field);
    }
  );
}
```

RESOLVING A FIELD WITH OUR RESOLVERREGISTRY

```
query {
    chatMessages(first: 1) {
        pageInfo {
            hasPreviousPage
            hasNextPage
            startCursor
            endCursor
        }
        edges {
            node {
                id
                sender {
                    id
                    displayName
                }
                sent {
                    timestamp
                }
                content {
                    ... on MediaChatMessageContent {
                        text
                    }
                }
            }
        }
    }
}
```




```
{
  "data": {
    "chatMessages": {
      "pageInfo": {
        "hasPreviousPage": false,
        "hasNextPage": true,
        "startCursor": “TRUNCATED",
        "endCursor": "TRUNCATED"
      },
      "edges": [
        {
          "node": {
            "id": "4bcb1252-e27d-4f2f-abd3-4f4f64cd2584",
            "sender": {
              "id": "c61e077f-f4c1-4b40-a403-869ee9a6e944",
              "displayName": "DrupalCon"
            },
            "sent": {
              "timestamp": "1649750295"
            },
            "content": {
              "text": "Welcome to DrupalCon!"
            }
          }
        }
      ]
    }
  }
}
```

WANT MORE?

Testing

Directives

Schema Design

Custom validation

QueryLoader Plugins

#graphql on drupal.slack.com

www.alexandervarwijk.com

twitter.com/Kingdutch

DrupalCon
PRAGUE2022
20-23 SEPTEMBER

Join us for contribution opportunities

20-23 September, 2022
Room C2 + C3

Mentored Contribution

23 September: 09:00 - 18:00
Room C2 + C3

First Time Contributor Workshop

20 September: 17:15 - 18:00
Room D9
21 September: 10:30 - 11:15
Room D9
23 September: 09:00 - 12:30
Room C2

General Contribution

20 - 22 September: 9:00 - 18:00
Room C3
23 September: 9:00 - 18:00
Room C2 + C3

#DrupalContributions

What did you think?

Please fill in this session survey directly from the Mobile App.

We appreciate your **feedback**!

Please take a moment to fill out:

1

the general
conference survey
Flash the QR code
OR
It will be sent by email

2

the **Individual session** surveys
(located under each session description)