Prebid Server Go Modularity

Technical Specification

Overview

Prebid Server allows **custom code execution** through predefined injection points while processing the requests. In order to add invocation code the **hook interfaces** must be implemented and properly tuned in application configuration. Single module may contain multiple hooks implemented for various endpoints.

Based on PRD - Prebid Server Modularity PRD

High-level plan

Go packages are used to modularize the application. The Prebid Server repository contains a package “modules” located in the root project directory. It includes all available PBS modules.

Here is the the new PBS project structure:

```
+- prebid-server/
  +- modules/ <- package with modules
    +- builder.go <- list of all available modules
    +- {YOUR_VENDOR_NAME}/ <- top-level package used to group modules
from the same vendor
      +- {YOUR_MODULE_NAME}/ <- package with module source code
        +- module.go <- file with module initialization function
```

The "modules" directory serves as the package housing various modules and their hook implementations. Inside this directory, we maintain a clear separation between modules belonging to different vendors, denoted by "{YOUR_VENDOR_NAME}," and within each vendor's package, we further isolate individual modules, identified by "{YOUR_MODULE_NAME}." Each module is contained within its own package, allowing for efficient distribution and usage. The central "builder.go" file serves as a comprehensive list of all available modules, enabling easy integration and configuration.

Implementation guideline

Package naming convention

Module directory names ({YOUR_VENDOR_NAME}/YOUR_MODULE_NAME/}) must consist of valid identifiers. A valid identifier is defined as a sequence of one or more letters, including an underscore character (_), and digits. All other symbols such as -, ., etc. are not permitted. {YOUR_VENDOR_NAME}/YOUR_MODULE_NAME/} should be unique enough to avoid collisions with the other modules.

Building modules

To ensure correct building and initialization of your module by the PBS module builder, which is located at modules/modules.go, your module must be enabled in the application config (described in the Switching modules on/off section below), must have a module.go file (as shown in the PBS project structure above) and include a builder method with the following signature:

```
ModuleBuilderFn func(
    cfg json.RawMessage,
    deps moduledeps.ModuleDeps,
) (interface{}, error)
```

Here's a breakdown of the parameters:

Parameter	**Description**
`cfg`	Module specific configuration, format: map[vendor_name]map[module_name]interface{}
`deps`	Dependencies that a module may require

The interface{} returned by the ModuleBuilderFn would be the Module type which implements at least one Hook interface. Details on hook interfaces will be described further in the

documentation. PBS uses type assertion to find out which hook interfaces are implemented by the module.

Here's an example of how to implement the module and its builder:

```
type Module struct{}

func Builder(_ json.RawMessage, _ moduledeps.ModuleDeps)
(interface{}, error) {
      return Module{}, nil
}

func (m Module) HandleBidderRequestHook(
      _ context.Context,
      _ hookstage.ModuleInvocationContext,
      _ hookstage.BidderRequestPayload,
) (hookstage.HookResult[hookstage.BidderRequestPayload], error) {
      // Implement your hook logic here
      ...
}
```

Exposing modules

All available modules are exposed through the modules/builder.go file. This file is auto-generated, so you shouldn't edit it manually.

```
func builders() ModuleBuilders {
      return ModuleBuilders{
            "{YOUR_VENDOR_NAME}": {
                  "{YOUR_MODULE_NAME}": ModuleBuilderFn,
            },
            ...
      }
}
```

To register a new module, you just need to run **one** of the following commands from the PBS root directory:

```
make build-modules
go generate modules/modules.go
```

This command(s) scans the modules/ directory for files matching the pattern modules/*/*/module.go and adds all matching packages to the modules/builder.go file.

Injecting dependencies

Modules may inject any of required dependencies that they may need for hook execution through ModuleDeps struct described at modules/moduledeps/deps.go which is passed as an argument to the modules builder function which is responsible for the module initialization.

Switching modules on/off

The functionality of modules within the Prebid Server can be selectively enabled or disabled to suit your specific requirements.

Global Module Control: To control the overall activation of modules, use the hooks.enabled property at the root of the global application configuration. By default, this follows an opt-in model, meaning that all modules are disabled by default unless explicitly enabled through configuration.

Individual Module Control: For finer-grained control, each module can be enabled or disabled individually. Locate the configuration property at **hooks.modules.VENDORNAME.MODULENAME.enabled**. If a module is disabled or the configuration is missing, it won't be initialized by the module's builder.

Here's an example of the prebid.yaml configuration file illustrating both global and individual module control:

```
external_url: '...'
host: '...'
port: '...'
# Other configuration settings...
hooks:
  enabled: true # by default initialized to false in Viper
  modules:
      foo:
        bar:
          enabled: true # by default initialized to false in Viper
```

Versioning

At the very first steps of PBS Modularity feature implementation it is hard to predict the convenient way of modules' versioning. At present, the PBS-Go core does not introduce a dedicated mechanism for handling separate module versions. Instead, we entrust the versioning of modules to their individual owners and developers.

Should any concerns or suggestions regarding module versioning arise, we encourage open discussion within the Prebid community. This collaborative approach ensures that versioning practices align with the needs and preferences of module creators and the wider PBS community.

Hook Interfaces and Hook Result

The Prebid Server processing workflow is segmented into various 'stages,' each allowing module authors to inject specific function signatures known as 'hooks.' To associate these hooks with their respective modules, PBS Core employs Go's duck typing ability. Each module can implement one or more of the seven available hook interfaces, each designed for a specific stage:

```
github.com/prebid/prebid-server/hooks/hookstage.Entrypoint
github.com/prebid/prebid-server/hooks/hookstage.RawAuctionRequest
github.com/prebid/prebid-server/hooks/hookstage.ProcessedAuctionRequest
github.com/prebid/prebid-server/hooks/hookstage.BidderRequest
github.com/prebid/prebid-server/hooks/hookstage.RawBidderResponse
github.com/prebid/prebid-server/hooks/hookstage.AllProcessedBidResponses
github.com/prebid/prebid-server/hooks/hookstage.AuctionResponse
```

While it's not mandatory for a module to implement all of these interfaces, at least one is required to define the module's functionality. Each of these interfaces is associated with a corresponding payload type designed to handle incoming data (e.g., EntrypointPayload, RawAuctionRequestPayload, ProcessedAuctionRequestPayload, etc.). These interfaces return a generic HookResult structure that encapsulates the outcome of executing a specific hook instance.

```
type HookResult[T any] struct {
     Reject        bool
     NbrCode       int
     Message       string
     ChangeSet     ChangeSet[T]
     Errors        []string
     Warnings      []string
     DebugMessages []string
     AnalyticsTags hookanalytics.Analytics
     ModuleContext ModuleContext
}
```

The HookResult structure comprises the following components:

Field	Description
`Reject`	Indicates whether the request should be rejected by the hook, resulting

	in the termination of the request processing
`NbrCode`	Specifies the reason code for a "no bid" response, which must be provided if the hook rejects the request
`Message`	Arbitrary message added by hook
`ChangeSet`	A set of changes that the module intends to apply to the hook payload in the form of mutations
`Errors`	A list of error messages, may be empty
`Warnings`	A list of warning messages, may be empty
`DebugMessages`	A list of debug messages, may be empty
`AnalyticsTags`	Analytics tags reported by this hook, optional
`ModuleContext`	Arbitrary data passed between module hooks at different stages

Hooks

Auction/Amp hooks

The hooks described in this section are available for the following endpoints: Auction (/openrtb2/auction) and Amp (/openrtb2/amp).

Entrypoint hook

Entrypoint hooks are invoked very early in the request processing pipeline and operate over the HTTP request and the raw body.

At this stage account configuration is not available yet. This is why entrypoint hooks could not have account-specific configuration and may be part of global hooks programs only.

If a rejection occurs at this stage, it results in sending an empty BidResponse with the NBR (No Bid Response) code indicating the reason for rejection.

Entrypoint hook types:

```
type Entrypoint interface {
     HandleEntrypointHook(
          context.Context,
          ModuleInvocationContext,
          EntrypointPayload,
     ) (HookResult[EntrypointPayload], error)
```

```
}

type EntrypointPayload struct {
      Request *http.Request
      Body    []byte
}
```

EntrypointPayload consists of an HTTP request and a raw body of the openrtb2.BidRequest. For amp endpoint the body is nil. Hooks are allowed to modify this data using mutations.

Field	Description
`Request`	HTTP request
`Body`	Raw body of the openrtb2.BidRequest

The ModuleInvocationContext will be utilized in every hook interface and contains data passed to the module hook during invocation. Additionally, there is a ModuleContext type nested within the ModuleInvocationContext, which can be employed to pass arbitrary data between different stages/hooks of the same module.

```
type ModuleInvocationContext struct {
      AccountConfig json.RawMessage
      Endpoint string
      ModuleContext ModuleContext
}

type ModuleContext map[string]interface{}
```

Field	Description
`AccountConfig`	Module configuration customized at the account level
`Endpoint`	Path of the current endpoint where the hook is being invoked
`ModuleContext`	Arbitrary data that can be exchanged between module hooks at various stages

Raw Auction Request hook

Raw auction request hooks are invoked **only** for */openrtb2/auction* endpoint after retrieving the account config but before the request is parsed and any additions are made like merging stored requests and injecting implicit parameters.

At this stage and all subsequent stages, the account configuration is available to every hook, enabling the configuration of account-level execution plans. This accessibility is provided because the account-level module configuration is passed to the hooks.

If a rejection occurs at this stage, it results in sending an empty BidResponse with the NBR (No Bid Response) code indicating the reason for rejection.

Raw Auction Request hook types:

```
type RawAuctionRequest interface {
     HandleRawAuctionHook(
          context.Context,
          ModuleInvocationContext,
          RawAuctionRequestPayload,
     ) (HookResult[RawAuctionRequestPayload], error)
}

type RawAuctionRequestPayload []byte
```

RawAuctionRequestPayload represents a raw body of the openrtb2.BidRequest. Hooks are allowed to modify the body using mutations.

Processed Auction Request hook

Processed auction request hooks are invoked after request is parsed and filled with stored requests and populated with implicit parameters.

If a rejection occurs at this stage, it results in sending an empty BidResponse with the NBR (No Bid Response) code indicating the reason for rejection.

Processed Auction Request hook types:

```
type ProcessedAuctionRequest interface {
     HandleProcessedAuctionHook(
          context.Context,
          ModuleInvocationContext,
          ProcessedAuctionRequestPayload,
     ) (HookResult[ProcessedAuctionRequestPayload], error)
}

type ProcessedAuctionRequestPayload struct {
     RequestWrapper *openrtb_ext.RequestWrapper
}
```

ProcessedAuctionRequestPayload consists of the openrtb_ext.RequestWrapper object. Hooks are allowed to modify openrtb_ext.RequestWrapper using mutations.

Field	Description
`RequestWrapper`	Wraps the OpenRTB request to provide a storage location for unmarshalled ext fields

Bidder Request hook

Bidder request hooks are invoked for each bidder participating in the auction and operate over the BidRequest instance distilled for the particular bidder.

If a rejection occurs at this stage, it results in skipping the particular bidder's request with the NBR (No Bid Response) code indicating the reason for rejection.

Bidder Request hook types:

```
type BidderRequest interface {
     HandleBidderRequestHook(
          context.Context,
          ModuleInvocationContext,
          BidderRequestPayload,
     ) (HookResult[BidderRequestPayload], error)
}

type BidderRequestPayload struct {
     Request *openrtb_ext.RequestWrapper
     Bidder  string
}
```

BidderRequestPayload consists of the openrtb2.BidRequest object distilled for the particular bidder. Hooks are allowed to modify openrtb2.BidRequest using mutations.

Field	Description
`Request`	Wraps the OpenRTB request to provide a storage location for unmarshalled ext fields
`Bidder`	Code of the bidder the hook is being invoked on

Raw Bidder Response hook

Raw bidder response hooks are invoked for each bidder participating in the auction and operate over a list of bids returned by a particular bidder.

If a rejection occurs at this stage, it results in ignoring the bidder's response with the NBR (No Bid Response) code indicating the reason for rejection.

Raw Bidder Response hook types:

```
type RawBidderResponse interface {
     HandleRawBidderResponseHook(
          context.Context,
          ModuleInvocationContext,
          RawBidderResponsePayload,
     ) (HookResult[RawBidderResponsePayload], error)
}

type RawBidderResponsePayload struct {
     Bids   []*adapters.TypedBid
     Bidder string
}
```

RawBidderResponsePayload consists of a list of bids returned by a particular bidder.

Field	Description
`Bids`	Bids returned by a particular bidder.
`Bidder`	Code of the bidder the hook is being invoked on

All Processed Bid Responses hook

All Processed Bid Responses hooks are invoked for over a list of all processed responses received from bidders before a winner is chosen.

Rejection is not allowed at this stage.

All Processed Bid Responses hook types:

```
type AllProcessedBidResponses interface {
     HandleAllProcessedBidResponsesHook(
          context.Context,
          ModuleInvocationContext,
          AllProcessedBidResponsesPayload,
     ) (HookResult[AllProcessedBidResponsesPayload], error)
}

type AllProcessedBidResponsesPayload struct {
     Responses map[openrtb_ext.BidderName]*entities.PbsOrtbSeatBid
}
```

AllProcessedBidResponsesPayload consists of a list of all processed responses received from bidders. Hooks are allowed to modify the payload object and discard bids using mutations.

Field	Description
`Responses`	SeatBid returned by an AdaptedBidder for a particular bidder

Auction Response hook

Auction response hooks are triggered at the conclusion of the request processing pipeline. It's worth noting that hooks at this stage are invoked even if the request was rejected in earlier stages, providing one last opportunity to modify the response.

Rejection is not allowed at this stage.

Auction response hook types:

```
type AuctionResponse interface {
      HandleAuctionResponseHook(
            context.Context,
            ModuleInvocationContext,
            AuctionResponsePayload,
      ) (HookResult[AuctionResponsePayload], error)
}

type AuctionResponsePayload struct {
      BidResponse *openrtb2.BidResponse
}
```

AuctionResponsePayload consists of a final openrtb2.BidResponse object that will be sent back to the requester. Hooks are allowed to modify openrtb2.BidResponse object.

Field	Description
`BidResponse`	Bid response populated with seat bids received from bidders and accompanying properties

Cookie Sync endpoint

The modules are not implemented for this endpoint.

Setuid endpoint

The modules are not implemented for this endpoint.

Event endpoint

The modules are not implemented for this endpoint.

Vtrack endpoint

The modules are not implemented for this endpoint.

Module Shared Data

Hooks have the capability to exchange specific information with other hooks within the same module through a shared data mechanism. However, it's important to note that this shared data is accessible only within hooks belonging to the same module.

The ModuleInvocationContext passed to each hook stage includes a ModuleContext field, represented as a map[string]interface{}, designed for transmitting arbitrary data between module hooks at different stages. The use of the interface{} type offers flexibility to module developers, allowing them to choose the data type best suited for their needs. It's essential to understand that PBS Core does not validate the logical consistency of this data but serves as a conduit for passing it between hooks within the module. To prevent potential synchronization issues, it's advisable for hook implementations to employ immutable data structures when sharing data.

```
type ModuleInvocationContext struct {
      AccountConfig json.RawMessage
      Endpoint string
      ModuleContext ModuleContext
}

type ModuleContext map[string]interface{}
```

Shared data is propagated to downstream hook implementations as part of the HookResult type, which represents the outcome of executing a specific hook instance. Notably, entrypoint hook implementations do not receive shared data at this stage, although they have the option to return it if needed.

Hook Repository

The HookRepository interface is used for convenient access to the hooks available through application.

In order to obtain the hook which should be invoked within the execution plan there is a method for searching the hook by id which is a composite value in the format: {vendor_name}.{module_name}

```
type HookRepository interface {
      GetEntrypointHook(id string) (hookstage.Entrypoint, bool)
      GetRawAuctionHook(id string)
          (hookstage.RawAuctionRequest, bool)
      GetProcessedAuctionHook(id string)
          (hookstage.ProcessedAuctionRequest, bool)
      GetBidderRequestHook(id string)
          (hookstage.BidderRequest, bool)
      GetRawBidderResponseHook(id string)
          (hookstage.RawBidderResponse, bool)
      GetAllProcessedBidResponsesHook(id string)
          (hookstage.AllProcessedBidResponses, bool)
      GetAuctionResponseHook(id string)
          (hookstage.AuctionResponse, bool)
}
```

Execution Plan Builder

The ExecutionPlanBuilder interface offers methods for organizing hooks into groups and sorting them in a predefined order, aligning with the intended execution plan for specific stages.

```
type ExecutionPlanBuilder interface {
      PlanForEntrypointStage(endpoint string)
          Plan[hookstage.Entrypoint]
      PlanForRawAuctionStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.RawAuctionRequest]
      PlanForProcessedAuctionStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.ProcessedAuctionRequest]
      PlanForBidderRequestStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.BidderRequest]
      PlanForRawBidderResponseStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.RawBidderResponse]
      PlanForAllProcessedBidResponsesStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.AllProcessedBidResponses]
```

```
      PlanForAuctionResponseStage(
          endpoint string,
          account *config.Account,
      ) Plan[hookstage.AuctionResponse]
}

type Plan[T any] []Group[T]

type Group[T any] struct {
      Timeout time.Duration
      Hooks []HookWrapper[T]
}

type HookWrapper[T any] struct {
      Module string
      Code string
      Hook T
}
```

In this context:

A Plan represents a collection of hook groups, each designated for a particular stage and organized in a specified order.

A Group constitutes a set of hooks sorted within that particular group. Each group is associated with a timeout field that defines the maximum duration, in milliseconds, within which the hooks in the group are allowed to run.

HookWrapper serves as a wrapper for individual hooks, encompassing essential meta-information such as the module name and hook code. The Module field contains the name of the module that provides the hook, following the format "vendor.module_name." The Code field represents an arbitrary value assigned to the hook within the hook execution plan, facilitating actions such as metrics reporting and debug logging.

A Hook represents a specific instance of a hook interface.

Hook invocation

Hooks are triggered based on the predefined execution plan defined in the configuration (for detailed information, please refer to the "Configuration" section) at various stages throughout the request processing workflow.

The responsibility of hook invocation is managed by a dedicated component with an interface resembling the following:

```
type HookStageExecutor interface {
      StageExecutor
      SetAccount(account *config.Account)
      GetOutcomes() []StageOutcome
}

type StageExecutor interface {
      ExecuteEntrypointStage(req *http.Request, body []byte)
([]byte, *RejectError)
      ExecuteRawAuctionStage(body []byte) ([]byte, *RejectError)
      ExecuteProcessedAuctionStage(req *openrtb_ext.RequestWrapper)
error
      ExecuteBidderRequestStage(req *openrtb2.BidRequest, bidder
string) *RejectError
      ExecuteRawBidderResponseStage(response
*adapters.BidderResponse, bidder string) *RejectError
      ExecuteAllProcessedBidResponsesStage(adapterBids
map[openrtb_ext.BidderName]*entities.PbsOrtbSeatBid)
      ExecuteAuctionResponseStage(response *openrtb2.BidResponse)
}

// hookExecutor implements both interfaces
type hookExecutor struct {
      account        *config.Account
      accountID      string
      endpoint       string
      planBuilder    hooks.ExecutionPlanBuilder
      stageOutcomes  []StageOutcome
      moduleContexts *moduleContexts
      metricEngine   metrics.MetricsEngine
      sync.Mutex
}
```

The HookStageExecutor operates at a higher level of processing, allowing for the initial setup of the account and retrieval of the final outcomes for all hooks. In contrast, the StageExecutor functions at a deeper level, where it executes the designated plan for each stage. It invokes hooks within each group as per the execution plan, prepares the results of hook invocations, manages hook mutations, gathers metadata for debugging and analytics purposes, tracks hook execution metrics, and assembles stage execution outcomes.

The next code defines data structures that capture the outcomes of executing various stages, groups of hooks within those stages, and individual hooks.

```
type StageOutcome struct {
      ExecutionTime
      Entity entity         `json:"entity"`
      Groups []GroupOutcome `json:"groups"`
      Stage  string         `json:"-"`
```

```
}

type GroupOutcome struct {
     ExecutionTime
     InvocationResults []HookOutcome `json:"invocation_results"`
}

type HookOutcome struct {
     ExecutionTime
     AnalyticsTags hookanalytics.Analytics `json:"analytics_tags"`
     HookID        HookID                  `json:"hook_id"`
     Status        Status                  `json:"status"`
     Action        Action   `json:"action"`
     Message       string   `json:"message"`
     DebugMessages []string `json:"debug_messages,omitempty"`
     Errors        []string `json:"-"`
     Warnings      []string `json:"-"`
}

type HookID struct {
     ModuleCode   string `json:"module_code"`
     HookImplCode string `json:"hook_impl_code"`
}

type ExecutionTime struct {
     ExecutionTimeMillis time.Duration
`json:"execution_time_millis,omitempty"`
}

type entity string

const (
     entityHttpRequest              entity = "http-request"
     entityAuctionRequest           entity = "auction-request"
     entityAuctionResponse          entity = "auction_response"
     entityAllProcessedBidResponses entity =
"all_processed_bid_responses"
)

type Status string

const (
     StatusSuccess          Status = "success"
     StatusTimeout          Status = "timeout"
     StatusFailure          Status = "failure"
     StatusExecutionFailure Status = "execution_failure"
)

type Action string

const (
```

```
      ActionUpdate Action = "update"
      ActionReject Action = "reject"
      ActionNone   Action = "no_action"
)
```

StageOutcome represents the result of executing a specific stage and aggregates information such as execution time, the type of processed entity, and outcomes for groups of hooks within that stage. The execution time for the stage is calculated as the sum of execution time of all its groups.

GroupOutcome encapsulates the outcome of executing a specific group of hooks within a stage. It tracks the execution time of the group and contains results for each individual hook within the group. Execution time is set to the longest execution time of its children.

HookOutcome provides detailed information about the result of executing a specific hook. It includes execution time (time of a specific hook without applying its result), analytics tags, a unique identifier for the hook, execution status, actions taken after hook execution (such as updates or rejections), arbitrary messages from the hook, debug messages, and any errors or warnings that may have occurred.

These data structures enable comprehensive tracking and reporting of hook execution outcomes, helping to monitor and manage the behavior of hooks within the Prebid Server's processing workflow.

Configuration

Default Module configuration

In PBS, the default approach is to refrain from providing modules with any predefined configuration. Instead, modules are granted full autonomy and responsibility for their own configuration.

To configure a module, you should place its configuration at the root level of the application configuration, using the following path: **hooks.modules.VENDORNAME.MODULENAME**. This configuration will subsequently be provided as an argument to the module builder, as detailed in the "Building Modules" section.

Here's an example of module configuration in the prebid.yaml file:

```
hooks:
  modules:
    sample-vendor:
```

```
      sample-module:
        enabled: true #required parameter for module initialization
        sample-param1: sample-value1
        sample-param2: sample-value2
```

Account Module configuration

This configuration describes hook-specific properties and contains of the next sections:

1. Account Hook configuration
Hooks may require runtime information or account-specific settings that cannot be accommodated within the application-level YAML configuration. Such options are passed to the ModuleInvocationContext during request processing, with the account details retrieved after the Entrypoint stage. It's important to note that the usage of this information is entirely at the discretion of the hook; PBS does not validate it but simply forwards the account configuration to the function. The path for the module's publisher account configuration is **hooks.modules.VENDORNAME.MODULENAME.**

Below is an example of module configuration at the account level in JSON format:

```
{
  ...
  "hooks": {
    "modules": {
      "sample-vendor": {
        "sample-module": {
          "number-key": 123,
          "string-key": "value",
          "object-key": {
            ...
          }
        }
      }
    }
  }
...
}
```

2. Account Hook execution plan.
See “Hook Execution Plan” section for details.

Hook Execution Plan

A hook execution plan consists of two parts:

- A host-level plan which is always executed and can not be skipped. It is defined at the host-company PBS configuration level. It's where the host company will invoke modules it requires for hosting operations.
- An account-level default execution plan can be defined by the PBS host company. This configuration can be overridden by account-config in the PBS database. So for example, a host company that services multiple accounts can define the default set of modules run across accounts, but then customize the modules used for certain accounts.

Each PBS host company will decide which modules it wants to enable and will set up an execution plan like this:

```
hooks:
  host_execution_plan: >
  {
    "endpoints": {
      "/openrtb2/auction": { # endpoint
        "stages": {
          "entrypoint": { # stage
            "groups": [
              {
                "timeout": 5,
                "hook_sequence": [
                  {
                    "module_code": "vendor1.module1",
                    "hook_impl_code": "hook1"
                  },
                  {
                    "module_code": "vendor2.module2",
                    "hook_impl_code": "hook2"
                  }
                ]
              }
            ]
          }
        }
      }
    }
  }
  default_account_execution_plan: >
  {
    "endpoints": {
      "/openrtb2/auction": { # endpoint
        "stages": {
          "entrypoint": { # stage
            "groups": [
              {
                "timeout": 5,
                "hook_sequence": [
                  {
```

```
                    "module_code": "vendor1.module1",
                    "hook_impl_code": "hook1"
                  },
                  {
                    "module_code": "vendor2.module2",
                    "hook_impl_code": "hook2"
                  }
                ]
              }
            ]
          }
        }
      }
    }
  }
```

An account-level execution plan has similar structure and is defined as follows in account configuration JSON:

```
{
  "hooks": {
    "execution_plan": {
      "Endpoints": {
        "/openrtb2/auction": { # endpoint
          "stages": {
            "entrypoint": { # stage
              "groups": [
                {
                  "timeout": 5,
                  "hook_sequence": [
                    {
                      "module_code": "vendor1.module1",
                      "hook_impl_code": "hook1"
                    },
                    {
                      "module_code": "vendor2.module2",
                      "hook_impl_code": "hook2"
                    }
                  ]
                }
              ]
            }
          }
        }
      }
    }
  }
}
```

The host-level plan is always executed first, then the resolved account-level plan is executed.

Within each stage, the order of groups is of paramount significance, as PBS executes these groups precisely as defined in the configuration. However, it's essential to note that the order of individual hooks within a group is not preserved. Each hook operates in its separate goroutine, which means their execution order within a group can vary. This distinction becomes particularly relevant when multiple modules need to interact with the same fields in the request. Hence, careful consideration should be given when designing and configuring modules to ensure seamless coordination when dealing with shared data in the request.

Endpoint and stage enums that appear in either configuration are defined as follows:

```
const (
      EndpointAuction = "/openrtb2/auction"
      EndpointAmp     = "/openrtb2/amp"
)

type Stage string

const (
      StageEntrypoint                Stage = "entrypoint"
      StageRawAuctionRequest         Stage = "raw_auction_request"
      StageProcessedAuctionRequest   Stage =
"processed_auction_request"
      StageBidderRequest             Stage = "bidder_request"
      StageRawBidderResponse         Stage = "raw_bidder_response"
      StageAllProcessedBidResponses Stage =
"all_processed_bid_responses"
      StageAuctionResponse           Stage = "auction_response"
)
```

Debug Output

Once all hooks have completed processing for the /openrtb2/auction and /openrtb2/amp endpoints, debug information may be incorporated into the output using the key **ext.prebid.modules**. PBS Core takes responsibility for determining whether debug mode is enabled, alleviating modules from this concern. Modules can always return warning, debug messages, and errors without requiring additional checks.

Three flags play a pivotal role in enabling debug output:

- **request.test** - Accepts values 0 or 1. It serves as an indicator of the test mode, where 0 denotes live mode, and 1 signifies test mode.
- **request.ext.prebid.debug** - Accepts values true or false, indicating whether debug mode is requested.

- **debug_allow** - Accepts values true or false, retrieved from the publisher account configuration, controlling debug mode.

Debug output is enabled if either request.test or request.ext.prebid.debug is set to true. However, it's important to note that the debug_allow flag in the account configuration must also match the values of the previous two flags. If the account disallows debug output, it will not be included in the response, even if request.test or request.ext.prebid.debug are true.

Additionally, there's an **ext.prebid.trace** flag with possible values of basic or verbose, responsible for providing trace information in the response under the key **ext.prebid.modules.trace**. The presence of the ext.prebid.modules.trace key is solely controlled by the ext.prebid.trace flag. It remains active even if debug mode is disabled. Below is an illustrative table demonstrating how these flags govern the details of output information:

Response Field	**Triggered By**	**Output**
response.ext.prebid.modules.errors.MODULECODE	request.ext.prebid.debug: true or request.test:1; and debug_allow: true	array of error strings from the module
response.ext.prebid.modules.warnings.MODULECODE	request.ext.prebid.debug: true or request.test:1; and debug_allow: true	array of warning strings from the module
response.ext.prebid.modules.trace	request.ext.prebid.trace: basic|verbose	see table below for which fields are in basic or verbose

Trace output:

response.ext.prebid.modules.*	**ext.prebid.trace = basic**	**ext.prebid.trace = verbose**
trace.*.executiontimemillis	+	+
trace.*.status	+	+
trace.*.message	+	+
trace.*.action	+	+
trace.*.analyticstag	+	+
trace.*.debugmessages		+

Here is an example of how debug output section looks like in response.ext for /openrtb2/auction endpoint:

```
{
  "prebid": {
    "modules": {
      "errors": {
        "vendor-1": {
          "module-1": [
            "error 1"
          ],
          "module-2": [
            "error 2",
            "error 4"
          ]
        },
        "vendor-2": {
          "module-3": [
            "error 3"
          ]
        }
      },
      "warnings": {
        "vendor-2": {
          "module-3": [
            "warning 1"
          ]
        }
      },
      "trace": {
        "execution_time_millis": 600,
        "stages": [
          {
            "stage": "entrypoint",
            "execution_time_millis": 300,
            "outcomes": [
              {
                "entity": "http-request",
                "execution_time_millis": 300,
                "groups": [
                  {
                    "execution_time_millis": 200,
                    "invocation_results": [
                      {
                        "hook_id": {
                          "module_code": "vendor-1.module-1",
                          "hook_impl_code": "foobar"
                        },
                        "status": "success",
                        "action": "update",
                        "execution_time_millis": 200,
                        "analytics_tags": {
```

```
                        "activities": [
                          {
                            "name": "device-id",
                            "status": "success",
                            "results": [
                              {
                                "status": "success-allow",
                                "values": {
                                  "foo": "bar"
                                },
                                "appliedto": {
                                  "impids": [
                                    "impId1"
                                  ],
                                  "request": true
                                }
                              }
                            ]
                          },
                          {
                            "name": "define-blocks",
                            "status": "error"
                          }
                        ]
                      },
                      "debug_messages": [
                        "debug message 1"
                      ],
                      "message": ""
                    },
                    {
                      "hook_id": {
                        "module_code": "vendor-1.module-2",
                        "hook_impl_code": "barfoo"
                      },
                      "status": "success",
                      "action": "update",
                      "execution_time_millis": 50,
                      "analytics_tags": {},
                      "message": ""
                    }
                  ]
                },
                {
                  "execution_time_millis": 100,
                  "invocation_results": [
                    {
                      "hook_id": {
                        "module_code": "vendor-2.module-3",
                        "hook_impl_code": "baz"
                      },
```

```
                    "status": "success",
                    "action": "update",
                    "execution_time_millis": 100,
                    "analytics_tags": {},
                    "debug_messages": [
                      "debug message 1",
                      "debug message 2"
                    ],
                    "message": ""
                  },
                ]
              }
            ]
          }
        ]
      },
      {
        "stage": "bidder_request",
        "execution_time_millis": 300,
        "outcomes": [
          {
            "entity": "adf",
            "execution_time_millis": 200,
            "groups": [
              {
                "execution_time_millis": 200,
                "invocation_results": [
                  {
                    "hook_id": {
                      "module_code": "vendor-1.module-1",
                      "hook_impl_code": "foobar"
                    },
                    "status": "success",
                    "action": "update",
                    "execution_time_millis": 200,
                    "analytics_tags": {},
                    "debug_messages": [
                      "debug message 1"
                    ],
                    "message": ""
                  }
                ]
              }
            ]
          },
          {
            "entity": "adot",
            "execution_time_millis": 300,
            "groups": [
              {
                "execution_time_millis": 200,
```

```
                    "invocation_results": [
                      {
                        "hook_id": {
                          "module_code": "vendor-1.module-2",
                          "hook_impl_code": "barfoo"
                        },
                        "status": "success",
                        "action": "no_action",
                        "execution_time_millis": 200,
                        "analytics_tags": {},
                        "message": ""
                      }
                    ]
                  }
                ]
              }
            ]
          }
        ]
      }
    }
  }
}
```


The AMP response has the same response structure but it is located under the key of **ortb2.ext.**

Metrics

As part of its processing workflow, Prebid Server takes on the responsibility of populating metrics. This metric population occurs during the handling of each hook response. PBS provides the capability to collect metrics and store them in two different databases: InfluxDB and Prometheus.

Global hook metrics

The naming format employed for global metrics stored in InfluxDB follows this structure:

modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.call
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.execution_error
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.timeout
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.failure

modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.success.noop
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.success.update
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.success.reject
modules.module.{VENDORNAME}_{MODULENAME}.stage.{STAGE}.duration

The naming format employed for global metrics stored in Prometheus follows this structure:

modules_{VENDORNAME}_{MODULENAME}_called
modules_{VENDORNAME}_{MODULENAME}_execution_errors
modules_{VENDORNAME}_{MODULENAME}_timeouts
modules_{VENDORNAME}_{MODULENAME}_failed
modules_{VENDORNAME}_{MODULENAME}_success_noops
modules_{VENDORNAME}_{MODULENAME}_success_updates
modules_{VENDORNAME}_{MODULENAME}_success_rejects
modules_{VENDORNAME}_{MODULENAME}_duration

The metric name in Prometheus does not include the stage name since it is retained as a metric label.

Account hook metrics

The account metrics will be shown if `**metrics.disabled_metrics.account_modules_metrics**` setting is enabled in the application configuration.

The naming format employed for account metrics stored in InfluxDB follows this structure:

account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.call
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.failure
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.success.noop
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.success.update

account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.success.reject
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.duration
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.timeout
account.{ACCOUNTID}.modules.module.{VENDORNAME}_{MODULENAME}.execution_error

PBS refrains from storing account metrics in Prometheus due to potential cost implications. Given the large number of accounts, querying and storing such metrics could lead to resource-intensive operations, making it impractical and expensive.

Analytics Tags

Hooks have the capability to report analytics tags, which PBS-Core subsequently passes to registered analytics adapters. These analytics tags can be included in the HookResult after the execution of a specific hook instance. When a hook reports such tags, they are stored in the HookOutcomes alongside other invocation details and are made accessible to analytics adapters. Analytics adapters receive these tags through the Auction/AMP analytic object.

It's important to note that the internal structure of analytics tags is custom and varies depending on the specific module's implementation. As a result, analytics modules that intend to report on specific behaviors must be coded to understand the structure of the tags associated with that module.

Here are the main types of Analytics Tags that can be utilized:

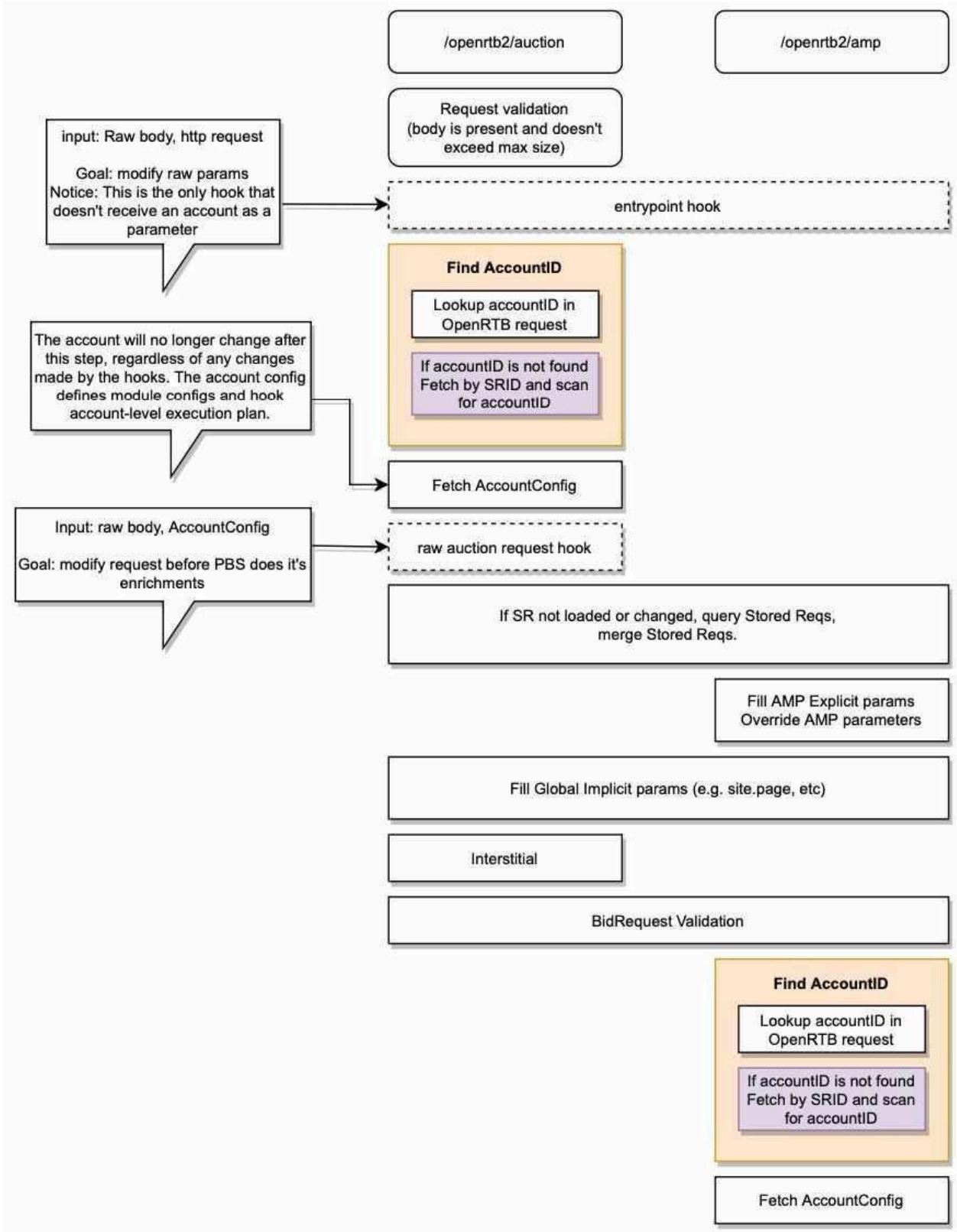
```
type Analytics struct {
      Activities []Activity `json:"activities,omitempty"`
}

type Activity struct {
      Name    string         `json:"name"`
      Status  ActivityStatus `json:"status"`
      Results []Result       `json:"results,omitempty"`
}

type ActivityStatus string

const (
      ActivityStatusSuccess ActivityStatus = "success"
      ActivityStatusError   ActivityStatus = "error"
)
```

```
type Result struct {
      Status    ResultStatus           `json:"status,omitempty"`
      Values    map[string]interface{} `json:"values,omitempty"`
      AppliedTo AppliedTo              `json:"appliedto,omitempty"`
}

type AppliedTo struct {
      Bidder   string   `json:"bidder,omitempty"`
      BidIds   []string `json:"bidids,omitempty"`
      ImpIds   []string `json:"impids,omitempty"`
      Request  bool     `json:"request,omitempty"`
      Response bool     `json:"response,omitempty"`
}

type ResultStatus string

const (
      ResultStatusAllow  ResultStatus = "success-allow"
      ResultStatusBlock  ResultStatus = "success-block"
      ResultStatusModify ResultStatus = "success-modify"
      ResultStatusError  ResultStatus = "error"
)
```

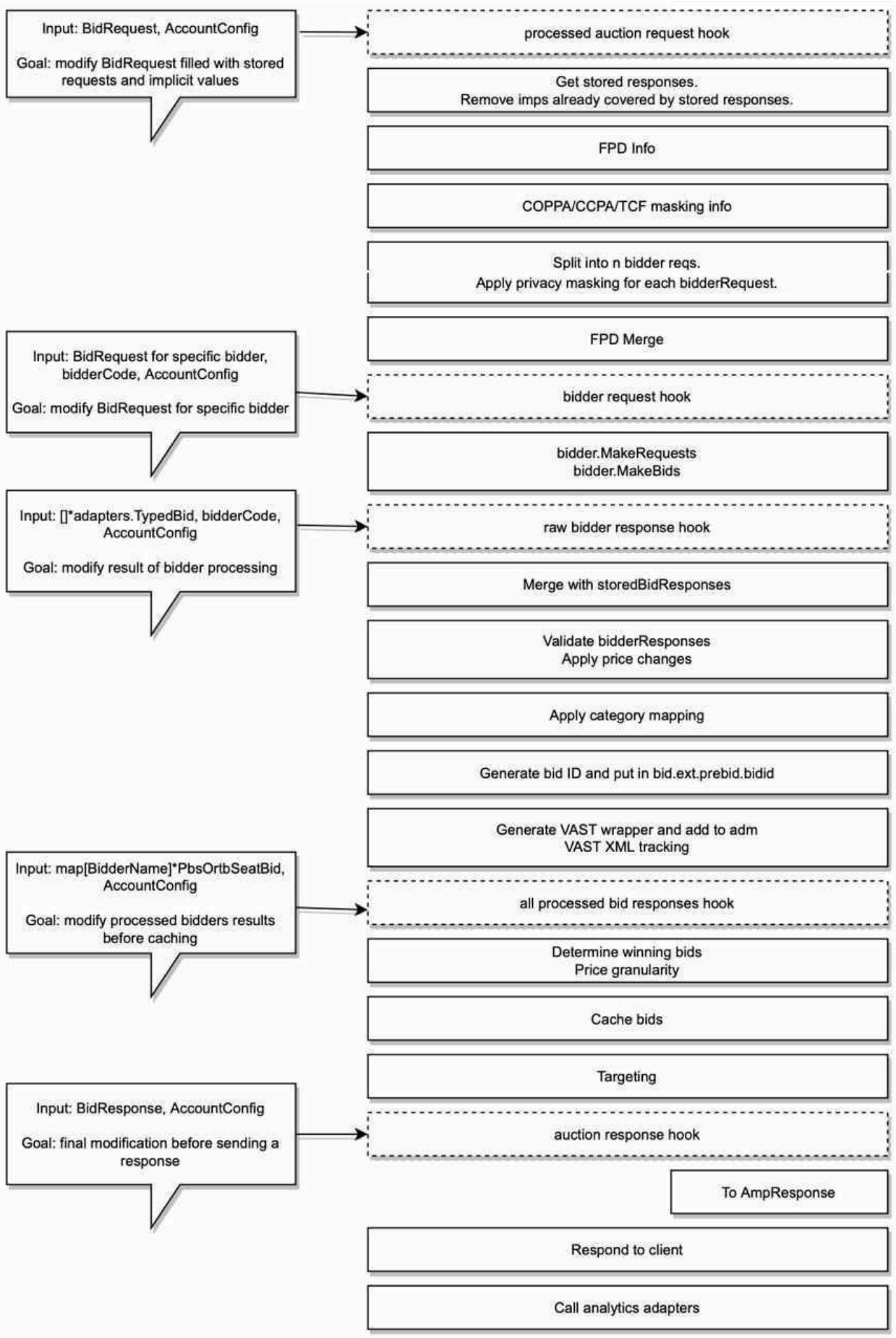
Appendix A - PBS-Go Hooks Location

This diagram provides a comprehensive overview of the specific actions performed by PBS-Core before and after the execution of each hook. It covers the /openrtb2/auction and /openrtb2/amp endpoints in precise detail, offering valuable insights into the sequence of operations at these stages.

PBS-Go Internal Flow Part 1



PBS-Go Internal Flow Part 2


Input: BidRequest, AccountConfig
Goal: modify BidRequest filled with stored requests and implicit values
processed auction request hook
Get stored responses.
Remove imps already covered by stored responses.
FPD Info
COPPA/CCPA/TCF masking info
Split into n bidder reqs.
Apply privacy masking for each bidderRequest.
FPD Merge
Input: BidRequest for specific bidder, bidderCode, AccountConfig
Goal: modify BidRequest for specific bidder
bidder request hook
bidder.MakeRequests
bidder.MakeBids
Input: []*adapters.TypedBid, bidderCode, AccountConfig
Goal: modify result of bidder processing
raw bidder response hook
Merge with storedBidResponses
Validate bidderResponses
Apply price changes
Apply category mapping
Generate bid ID and put in bid.ext.prebid.bidid
Generate VAST wrapper and add to adm
VAST XML tracking
Input: map[BidderName]*PbsOrtbSeatBid, AccountConfig
Goal: modify processed bidders results before caching
all processed bid responses hook
Determine winning bids
Price granularity
Cache bids
Targeting
Input: BidResponse, AccountConfig
Goal: final modification before sending a response
auction response hook
To AmpResponse
Respond to client
Call analytics adapters